

FONTEDIT

*Fontedit*

FONTEDIT

FONTEDIT

FONTEDIT

**Fontedit**

With FONTEDIT you can design your own character sets (or *fonts*) for the ATARI. For example, you might create a Russian alphabet, an APL character set, or special-purpose graphic symbols. After you design your character set, you can save a permanent copy on disk or cassette tape for later use. (We include a separate subroutine that you can use in your own programs to load a custom font from either disk or tape.)

Best of all, with FONTEDIT it is really easy to create a new character set (or just modify the existing ATARI characters). FONTEDIT shows you an enlarged copy of the character that you are designing. You move a cursor around within the big picture of the character, and turn the individual dots *on* or *off* with the joystick

button. In addition to seeing a large version of the character, you are shown a group of the characters together, as well as the entire character set as currently defined. For ease of use, each of the FONTEDIT commands are shown in a menu on the right-hand portion of the screen. FONTEDIT is friendly and easy to use: just grab a joystick and become a font designer! (Requires 16K memory with cassette, 24K with disk.)

KNOTWORK lets you design patterns of *Celtic interlace* (a technique used by 7th century Irish monks to illuminate manuscripts). When KNOTWORK is first run, it covers the screen of the ATARI with an unadorned knotwork pattern. You use the joystick to move an

arrow around on the screen. When you press the button on the joystick, a small vertical or horizontal line appears (called a *break* by the monks). It sounds simple, and it is! With some creativity, you can produce beautiful patterns that look like woven rope. Naturally, you are able to save a copy of the screen on tape or disk, and then load it back later. As you might expect, KNOTWORK uses custom graphics characters that were originally designed with FONTEDIT. (Requires 16K with cassette, 24K with disk.)



**IRIDIS** <sup>TM</sup> **2**





# IRIDIS 2

## Fontedit

Instructions . . . . .	2
Program Listing . . . . .	6
How It Works . . . . .	9

## Knotwork

Instructions . . . . .	22
Program Listing . . . . .	24
How It Works . . . . .	27

## Interlacing Without Erasing

Knotwork By Hand . . . . .	28
----------------------------	----

## Hacker's Delight

Useful Memory Locations . . . . .	36
-----------------------------------	----

## Loadfont

Subroutine to Load a Font . . . . .	46
-------------------------------------	----

## Oddments

Facts, Fancies and Rumors . . . . .	49
-------------------------------------	----

## The Oracle

Sees All, Knows All, Tells All . . . . .	51
--	----

## Novice Notes

About Bytes and Bits . . . . .	52
--------------------------------	----

## About IRIDIS Listings

How We Print the Unprintable . . . . .	56
--	----

Fontedit is a program which lets you design your own character sets (or **fonts**, as the printers call them) for the Atari. The redesigning can range from merely touching up the current characters to replacing everything with characters of your own design (an APL character set, for example). Once designed, the character sets can be saved for later use.

Some background is in order. The characters that you see on the screen of the Atari are made up of small dots. The dots are arranged in an 8x8 square, for a total of 64 dots. The characters are shown by making some of the dots visible, and keeping the rest hidden. The characters don't appear to be made of dots because the dots overlap, filling any holes that might have appeared. (Take a look at the table on page 56 which shows how we print Atari's characters; the dot patterns all show up clearly there.) What Fontedit does is let you change which dots can or can't be seen for any given character. By changing the dot pattern, you can change how the character looks. (Consider it electronic needlepoint, if you like.)

A further complication in the dot patterns is that some dots are green, and others are purple. (That is, with a black background. Other color backgrounds produce other color dots.) Specifically, four of the columns of dots are green, and the other four are purple. The colors alternate across the character. If a green and a purple dot touch, they combine into white. If you like, you use the colored dots to make characters that are all one color or the other, or a mixture of both.

Whenever Fontedit is awaiting your direction, it shows a blinking cursor on the screen. The cursor shows where the computer's attention is on the screen. The cursor blinks just to call your attention to it. You can move the cursor by pushing the joystick in the appropriate direction: push the joystick up to move the cursor up; push it right to move the cursor to the right; and so forth. (The diagonals work, too.) The cursor will keep moving until you let go of the joystick. If the cursor goes off one side of the picture, it will reappear on the other side.

When the cursor is blinking, you can also give one of Fontedit's commands. Find the key on the keyboard with the proper character on it (the character you see in the menu), and press it. Fontedit is watching the keyboard at the same time it's watching the joystick, so it responds to a keypress as quickly as it responds to the joystick. (Note that you need only press the one key. You don't need to press {CTRL} or {SHIFT} to give a command.)

To start, load Fontedit and type **RUN**. It will spend a little time setting up, and then present you with a menu of commands, as well as several displays of characters. The upper left display (an 8x8 square of dots) shows the character that you're currently editing, but blown up, so you can easily see the individual dots making it up. The solitary character to the right of the 8x8 square shows what the character looks like alone. The 5x5 set of characters below that shows what the character looks like repeated several times. Below the 8x8 square, the entire character set is shown at once, so you can get a feel for how the various characters look together. (The menu helps there, too.) When it starts up, Fontedit automatically "types" an Edit command for you, as it has no way of knowing which character you want to edit first.

Before you can edit a character, you must tell Fontedit that you want to edit that character. To do that, you must give the Edit command. (Except when Fontedit first starts up. Then it pretends that you gave one already.) When Fontedit is given the Edit command, it moves the cursor from the 8x8 square into the character set. (In



the character set, the cursor is a blinking square.) Use the joystick to move the cursor to the character you want to edit, and press the button. That character will be copied into the 8x8 square, and into the normal-sized samples beside the 8x8 square. The cursor is moved back to the 8x8 square, and you can begin editing the character using the joystick. If you change your mind about switching characters, press any key (except {BREAK}) on the keyboard. The cursor will move back up to the 8x8 square, and everything will be as if you had never given the Edit command. (The key you press will not be taken as a command.)

When you change characters using Edit, the pattern that the Undo command (described below) restores is changed. It becomes the current pattern for the character you selected, as it is at the time you select it. This is true even if you don't move the cursor. The only way to prevent changing Undo's pattern is to abort the Edit by pressing a key. Pressing a key to end Edit leaves things exactly as they would be had you never given Edit at all.

Once you've selected a character to edit, the cursor moves back into the 8x8 square. In the square, you will see one of the dots blinking. The cursor is where the blinking dot is. Use the joystick to move the cursor about the square. Once the cursor is at a dot you want to change, press the joystick button. If the dot is black, it will turn white; if it's white, it will turn black. By moving the cursor around and pressing the button, you can produce any pattern you like. When the dot pattern is the way you want it, just go off and do something else. While you're changing the 8x8 square, Fontedit is making corresponding changes to the character itself, so that what you see on the screen (in the two displays of the character to the right of the 8x8 square) is always up to date.

---

### **Summary of Fontedit Commands**

- A      *Restore current character to Atari's pattern for it.*
- C      *Copy another character's pattern into the current character's.*
- E      *Begin editing a different character's pattern.*
- F      *List the fonts saved on disk.*
- L      *Load a new font from disk or cassette.*
- Q      *Quit Fontedit, leaving the edited font in use.*
- R      *Reverse black and white in the pattern.*
- S      *Save the current font onto disk or cassette.*
- U      *Undo any changes made to a pattern since an Edit or a Load.*
- {CLEAR} *Make the pattern all black.*
- {UP}    *Roll the pattern up one row.*
- {DOWN} *Roll the pattern down one row.*
- {LEFT} *Roll the pattern left one column.*
- {RIGHT} *Roll the pattern right one column.*
- Joystick *Move the cursor*
- Button *Change the color of a dot, or  
         Select a character to Edit or Copy*

Fontedit has a number of commands besides Edit and character editing:

Reverse switches white and black throughout the pattern. All the white dots become black, and all the black dots become white. (The command name comes from computer parlance, in which we say that the character has been made "reverse video".)

Clear (given by pressing the {CLEAR} key) makes all the dots in the current character black, so that you can design your own character on a clean slate.

The four Roll commands (given by pressing the arrow keys) let you adjust the position of the character in the 8x8 square. Roll Left moves every dot in the character left by one dot's width. The dots that get pushed off the left side reappear at the right. Roll Right is the same as Roll Left, except the dots move to the right. Roll Up moves every dot up one dot's height. The dots that fall off the top reappear at the bottom. Roll Down is the same as Roll Up, except that the dots move down instead of up.

Finally, Copy lets you make the character look just like another character. If you like, you can leave it at that, or you can then modify the character. The technique of copying characters can often save work when you're making a character that looks a lot like another one (such as E and F).

After you press C, the cursor will move down into the character set, on top of the character being edited. Move it to the character you want to copy using the joystick, and press the button. That character's dot pattern will be copied into the current character's pattern. If you press a key on the keyboard instead of pressing the joystick button, the copy won't be done, and the original pattern will remain untouched. (The key you press will not be taken as a command.) In either case, the cursor moves back to the 8x8 square, and you can resume editing.

Occasionally, even the best of us make mistakes. Fontedit has two commands which permit you to change your mind, and go back to an older version of the character. Undo restores the dot pattern that the character had when you last Edited it, or when a font was last Loaded, whichever was most recent. Atari changes the pattern of the current character to the one that the Atari uses when you turn the computer on. The Atari command affects only the current character. (Note that Undo will even undo an Atari command.)

When all the characters are as you want them (or if you are rudely interrupted at your work), you can save the font in a file using the Save command. Save will ask for a file name to save the font as. Type in an eight (or less) letter name. You may give an extender if you like; Fontedit will use ".FNT" if you don't. If you have a disk, Fontedit will save the font there; otherwise it will use the cassette. (If you want to save to cassette when you have a disk, save to "C:".) When Fontedit is done saving the font, it will say "SAVE COMPLETE", and let you go back to playing with the font.

To get a font back again, use the Load command. Load works just like Save, except that it copies a font into the Atari, instead of out of it. (As with Save, loading from "C:" will make it read a cassette.) When it's done loading the font, it will say "LOAD COMPLETE".

If you've forgotten what the name of your font is, fear not. Fontedit can show you what fonts you've saved on the disk. (Cassettes have no file names or directories, so Fontedit can't help those of you without disks.) After you give the Font List command, Fontedit will ask you which disk drive you want to look on. Reply with a number from 1 to 4. Fontedit will fill the area at the bottom of the screen with font names. (It will only show you those fonts with an extender of ".FNT". If you have any with other extenders, they won't be shown, even if they are font files.)



When the screen is full, you will be told "PRESS RETURN FOR MORE". When you press a key (not just {RETURN}, actually), the bottom of the screen will be filled with more names of fonts. When all the font names have been shown, Fontedit will tell you "PRESS RETURN TO CONTINUE" editing. Again, any key will do. The bottom of the screen will be erased, and you can go back to editing characters.

Sooner or later, you'll realize that other things must be done. (Fontedit is a wonderful way to kill time; we speak from experience.) To leave Fontedit, give the Quit command. Fontedit will clear the screen, and give you back the normal Atari cursor. Your modified font will be left in effect, so that anything printed will show up in your characters. (Amaze your friends: list your program after you've loaded a strange font.)

After you leave Fontedit, several things can restore the normal Atari font instead of yours. Some can be counteracted, a few require rerunning Fontedit. Naturally, turning the Atari off wipes your font out of the machine (but not out of any files it's saved in). Also, pressing {RESET} will make the Atari go back to its own font. The only way to get back to your font is to rerun Fontedit. The Atari will also use its own font after you enter (and leave) the DOS, or change graphics modes. In this case, you can get your font back by typing

POKE 756, PEEK(106)+1

If you WANT the normal Atari font back again, pressing {RESET} is the simplest way to get it. If you want to edit the normal font, press {RESET}, then type **RUN**.

A warning about split-screen windows used with private fonts: when the Atari scrolls the text window, it scrolls 24 lines instead of just the four in the window. This results in part of your font being scrolled. As a result, private fonts will work with text windows only if the text window is never scrolled. It seems that the people who wrote the screen handler assumed that there would never be anything in memory after the screen memory. As a result, they felt free to overwrite the "nonexistent" memory after the screen. Unfortunately, the current font is kept after the screen memory. It is therefore subject to the ravages of the screen handler. (Also, we may not have found all the ways that screen diddling can go past the end of the screen internally. GRAPHICS 0 has worked quite reliably. Other graphics modes we don't have much experience with.)

If you prefer the screen to be a color other than black, you can change it by changing line 1060. Where it says 'SETCOLOR C2,C0,C0', change the first C0 to the number for the color you want. You can also change the second C0, to alter the brightness of the screen. Black was chosen for the greatest contrast between dots on and off.

--

Included with the programs in Iridis 2 are a few fonts that we have created. FANCY.FNT is basically the normal Atari font. However, it has been made fancier and has been generally spiffed up. (All that does not make it a 'has-been' font.) Anything you can use FANCY for, you can use the normal Atari font for, but FANCY looks better. (It is sometimes harder to read, due to the limitations of fitting each character into an 8x8 square.)

KNOTWORK.FNT is a special-purpose font used by the Knotwork program. It contains special characters that are used for making the knotwork patterns, instead of lower case and graphics. It is an example of using a font for something other than letters and text. KNOTWORK can be used only by the Knotwork program, but using it makes the Knotwork program much simpler than it would have been otherwise.

COMPUTER.FNT is a variant of the Computer typeface (which in turn was based on the MICR numbers that appear on the bottom of checks). The changes, as with FANCY, were made from constraints on fit (the 8x8 square again) or because we thought some characters looked better the way we did them. You can use this one to impress friends, or some such thing.

## Fontedit Listing

```

0 REM FONTEDIT
1 REM COPYRIGHT(C) 1980 THE CODE WORKS
2 REM BOX 550, GOLETA, CA. 93017
3 REM ALL RIGHTS RESERVED
10 REM
90 GOSUB 30000
100 C0=0:C1=1:C2=2:C4=4:C6=6:C7=7:C8=8
110 C9=9:C15=15:C18=18:C20=20:C23=23:C64=64
120 C96=96:C128=128:C255=255:C256=256
130 RAMTOP=106:ERRSAV=195:LOCK=702:CRSINH=752:CHBAS=756:KEYCODE=764:LMARG=82:
    ATRACT=77:TOPSCR=88:HATABS=794
135 CONSOL=53279
140 ABORT=155:POKE LMARG,C2
200 DIM STATE(C7,C7),BIT(C7),PRISTINE(C7)
210 DIM CMD$(C23),DOT$(C4),FILES$(C20),C$(C18)
300 DOT$=" {CLEAR}_ {BELL}":CMD$="ACEFLQRSU{CLEAR}<{UP DOWN LEFT RIGHT}==+*_|\"
310 T=C128:FOR I=C0 TO C7:BIT(I)=T:T=T/C2:NEXT I
320 HASDOS=C0
330 FOR I=C0 TO 12
340 IF PEEK(HATABS+3*I)=ASC("D") THEN HASDOS=C1
350 NEXT I
1000 POKE CHBAS,224:EOM=PEEK(RAMTOP)
1010 CHGEN=EOM*C256:T=C255-PEEK(CHGEN):POKE CHGEN,T
1020 IF PEEK(CHGEN)=T THEN EOM=EOM+C1:POKE CHGEN,C255-T:GOTO 1010
1030 EOM=EOM-C4:CHGEN=EOM*C256
1040 STD=(PEEK(1536)<>EOM) OR (PEEK(RAMTOP)>=EOM)
1060 POKE RAMTOP,EOM-C1:GRAPHICS C0:SETCOLOR C2,C0,C0
1070 IF STD=0 THEN POKE CHBAS,PEEK(1536):GOTO 1150
1080 I=USR(ADR("hh{E}Uh{E}Th{E}Wh{E}V{D}_{,}lT{Q}VHPyfuWJp{.}"),224*C256,CHGEN)
1090 RESTORE 1400:N=CHGEN+125*C8:FOR I=C0 TO C7:READ T:POKE N+I,T:NEXT I
1100 POKE EOM-C1,C0
1150 POKE CHBAS,EOM
1160 POKE CRSINH,C1
1170 FOR I=C0 TO C7:FOR J=C0 TO C7:POSITION I+C2,J:PRINT "{ESC CLEAR}";:NEXT J:NEXT I
1200 POSITION C2,C9
1210 FOR I=C0 TO C7:FOR J=C0 TO C15:PRINT "{ESC}";CHR$(I*16+J);:NEXT J:PRINT :NEXT I
1250 T=C0:RESTORE
1260 READ C$,FILES$:IF C$="" THEN 1500
1270 POSITION 27,T:PRINT C$;"|";FILES$
1280 T=T+C1:GOTO 1260
1300 DATA A,ATARI,C,COPY,E,EDIT,F,FONT LIST,L,LOAD FONT,Q,QUIT,R,REVERSE,S,SAVE FONT,U,UNDO
1310 DATA {ESC UP},ROLL UP,{ESC DOWN},ROLL DOWN
1320 DATA {ESC LEFT},ROLL LEFT,{ESC RIGHT},ROLL RIGHT,{4 LEFT}CLEAR,ALL DARK
1330 DATA {ESC} ,,{7 LEFT}JOYSTICK,TO MOVE,{5 LEFT}BUTTON,OFF{ESC LEFT ESC RIGHT}ON
1340 DATA ,
1400 DATA 0,0,0,24,24,0,0,0
1500 OLDSTICK=C15:STICKWAIT=C0:OLDTRIG=C1
1510 R=C0:C=C0:LET ONOFF=C1
1520 CHAR=ASC("@"):HOME=PEEK(TOPSCR)+C256*PEEK(TOPSCR+C1)
1530 GOSUB 9100:GOSUB 9000
1540 FOR I=C0 TO C7:PRISTINE(I)=PEEK(CURRENT+I):NEXT I
1550 GOTO 1900
1800 FOR I=C1 TO 1000:NEXT I
1810 POSITION C0,C23
1820 FOR I=C18 TO C23:PRINT "{DELETE-LINE UP}";:NEXT I
1900 BLINKWAIT=C0
2000 RVS=STATE(R,C)
2010 T=STICK(C0):IF T=C15 THEN OLDSTICK=T:GOTO 2300
2020 IF OLDSTICK>C15 THEN STICKWAIT=STICKWAIT-C1:IF STICKWAIT>C0 THEN 2300
2030 OLDSTICK=T:STICKWAIT=3
2040 T=C2+RVS+RVS:POSITION C+C2,R:PRINT "{ESC}";DOT$(T,T)
2050 POKE CONSOL,C0
2100 ON OLDSTICK+C1 GOSUB -2200,2200,2200,2200,2200,2200,2240,2220,2230,2200,2260,2280,
    2270,2200,2250,2210,2200

```



```

2110 IF R<C0 THEN R=C7
2120 IF R>C7 THEN R=C0
2130 IF C<C0 THEN C=C7
2140 IF C>C7 THEN C=C0
2150 LET ONOFF=C1:BLINKWAIT=C0
2170 POKE ATTRACT,C0:GOTO 2300
2200 RETURN
2210 R=R-C1:RETURN
2220 R=R-C1:C=C+C1:RETURN
2230 C=C+C1:RETURN
2240 R=R+C1:C=C+C1:RETURN
2250 R=R+C1:RETURN
2260 R=R+C1:C=C-C1:RETURN
2270 C=C-C1:RETURN
2280 R=R-C1:C=C-C1:RETURN
2300 T=STRIG(C0):IF T=OLDTRIG THEN 2400
2310 OLDTRIG=T:IF T=C1 THEN 2400
2320 RVS=C1-STATE(R,C):STATE(R,C)=RVS
2330 IF RVS=C0 THEN POKE CURRENT+R,PEEK(CURRENT+R)-BIT(C):GOTO 2350
2340 POKE CURRENT+R,PEEK(CURRENT+R)+BIT(C)
2350 T=C1+RVS+RVS+ONOFF:POSITION C+C2,R:PRINT "{ESC}";DOT$(T,T)
2400 IF PEEK(KEYCODE)<>C255 THEN 3000
2410 BLINKWAIT=BLINKWAIT-C1:IF BLINKWAIT>C0 THEN 2000
2420 LET ONOFF=C1-ONOFF:RVS=STATE(R,C)
2430 T=C1+RVS+RVS+ONOFF:POSITION C+C2,R:PRINT "{ESC}";DOT$(T,T)
2440 BLINKWAIT=C6:GOTO 2000
3000 T=C2+RVS+RVS:POSITION C+C2,R:PRINT "{ESC}";DOT$(T,T)
3010 GOSUB 9300:CMD=T
3050 IF CMD>C96 AND CMD<123 THEN CMD=CMD-32
3060 C$=CHR$(CMD)
3070 FOR I=C1 TO LEN(CMD$)
3080 IF CMD$(I,I)=C$ THEN 3100
3090 NEXT I:GOTO 2000
3100 IF I<13 THEN ON I GOTO 4100,4000,3400,4800,4400,9900,3300,4200,3500,3200,3200,3600
3110 ON I-12 GOTO 3700,3800,3900,3600,3700,3800,3900,3600,3700,3800,3900
3200 T=CHGEN+POSN:FOR I=C0 TO C7:POKE T+I,C0:NEXT I
3210 GOSUB 9000:GOTO 2000
3300 FOR I=C0 TO C7:POKE CURRENT+I,C255-PEEK(CURRENT+I):NEXT I
3310 GOSUB 9000:GOTO 2000
3400 GOSUB 8000:IF CH<C0 THEN 2000
3420 CHAR=CH
3430 GOSUB 9100:GOSUB 9000
3440 FOR I=C0 TO C7:PRISTINE(I)=PEEK(CURRENT+I):NEXT I
3450 R=C0:C=C0:GOTO 2000
3500 FOR I=C0 TO C7:POKE CURRENT+I,PRISTINE(I):NEXT I
3510 GOSUB 9000:GOTO 2000
3600 T=PEEK(CURRENT)
3610 FOR I=C0 TO C6:POKE CURRENT+I,PEEK(CURRENT+I+C1):NEXT I
3620 POKE CURRENT+C7,T
3630 GOSUB 9000:GOTO 2000
3700 T=PEEK(CURRENT+C7)
3710 FOR I=C6 TO C0 STEP -C1:POKE CURRENT+I+C1,PEEK(CURRENT+I):NEXT I
3720 POKE CURRENT,T
3730 GOSUB 9000:GOTO 2000
3800 FOR I=C0 TO C7
3810 T=PEEK(CURRENT+I)
3820 T=T*C2:IF T>C255 THEN T=T-C256+C1
3830 POKE CURRENT+I,T:NEXT I
3840 GOSUB 9000:GOTO 2000
3900 FOR I=C0 TO C7
3910 T=PEEK(CURRENT+I)
3920 T=T/C2:IF T<>INT(T) THEN T=INT(T)+C128
3930 POKE CURRENT+I,T:NEXT I
3940 GOSUB 9000:GOTO 2000
4000 GOSUB 8000:IF CH<C0 THEN 2000
4030 IF CH<C96 THEN CH=CH-32:IF CH<C0 THEN CH=CH+C96
4040 T=CHGEN+CH*C8
4050 FOR I=C0 TO C7:POKE CURRENT+I,PEEK(T+I):NEXT I
4060 GOSUB 9000:GOTO 2000
4100 T=224*C256+POSN
4110 FOR I=C0 TO C7:POKE CURRENT+I,PEEK(T+I):NEXT I
4120 GOSUB 9000:GOTO 2000
4200 C$="NAME TO SAVE AS":GOSUB 4600:IF FILE$="" THEN 2000
4210 TRAP 4300
4220 OPEN #C1,C8,C0,FILE$
4230 PRINT " SAVING AS ";FILE$;
4240 FOR I=CHGEN TO CHGEN+1023
4250 T=PEEK(I):IF I>CHGEN+C7 THEN POKE I,C255-T

```

```

4260 PUT #C1,T:POKE I,T:NEXT I:PRINT "{DELETE-LINE}      SAVE COMPLETE{BELL}";
4270 CLOSE #C1:GOSUB 9200:FOR I=C1 TO 150:NEXT I:PRINT "{DELETE-LINE}":GOTO 2000
4300 PRINT "[DELETE-LINE]CAN'T SAVE IN ";FILES$;"{BELL}":PRINT "ERROR ";PEEK(ERRSAV)
4310 CLOSE #C1:GOSUB 9200
4320 GOTO 1800
4400 C$="FONT TO LOAD FROM":GOSUB 4600:IF FILES$="" THEN 2000
4410 TRAP 4500
4420 OPEN #C1,C4,C0,FILES
4430 PRINT "      LOADING FROM ";FILES$;
4440 FOR I=CHGEN TO CHGEN+1023
4450 IF I>CHGEN+C7 THEN POKE I,C255-PEEK(I)
4460 GET #C1,T:POKE I,T:NEXT I:GOSUB 9000:POSITION C0,C20:
      PRINT "{DELETE-LINE}      LOAD COMPLETE{BELL}";
4470 CLOSE #C1:GOSUB 9200:FOR I=C1 TO 150:NEXT I:PRINT "{DELETE-LINE}"
4480 FOR I=C0 TO C7:PRISTINE(I)=PEEK(CURRENT+I):NEXT I:GOTO 2000
4500 PRINT "[DELETE-LINE]CAN'T LOAD FROM ";FILES$;"{BELL}":PRINT "ERROR ";PEEK(ERRSAV)
4510 CLOSE #C1:POKE EOM-C1,C0
4520 GOTO 1800
4600 T=PEEK(LOCK):POKE LOCK,C64
4610 POSITION C4,C20:PRINT C$;:INPUT C$
4620 PRINT "{UP DELETE-LINE}";:POKE LOCK,T
4630 FILES=C$:IF C$="" THEN RETURN
4640 FOR I=C1 TO LEN(C$)
4650 IF C$(I,I)=". " THEN 4700
4660 NEXT I
4670 FILES="D:":IF HASDOS=C0 THEN FILES="C:"
4680 FILES(3)=C$
4700 FOR I=C1 TO LEN(FILES)
4710 IF FILES(I,I)=". " THEN RETURN
4720 NEXT I
4730 FILES(LEN(FILES)+C1)=".FNT"
4740 RETURN
4800 POSITION C4,C20:IF HASDOS=C0 THEN PRINT "I CAN'T TELL WHAT'S ON THE CASSETTE.":GOTO 1800
4810 PRINT "WHICH DRIVE (1-4)?":GET #C7,T:PRINT "{DELETE-LINE}":IF T=ABORT THEN 2000
4820 T=T-ASC("0"):IF T<C1 OR T>C4 THEN POSITION C4,C20:PRINT "NO SUCH DRIVE.{BELL}":GOTO 1800
4830 FILES="D?*.FNT":FILES(C2,C2)=CHR$(T+ASC("0"))
4840 TRAP 4850:OPEN #C1,C6,C0,FILES:GOTO 4900
4850 POSITION C4,C20:PRINT "CAN'T READ DIRECTORY.":CLOSE #C1:GOTO 1800
4900 INPUT #C1,C$
4910 FOR I=C18 TO 21
4920 FOR J=C2 TO 29 STEP C9
4930 IF C$(C1,C1)<>" " THEN 5000
4940 POSITION J,I:PRINT C$(3,10)
4950 INPUT #C1,C$
4960 NEXT J:NEXT I
4965 IF C$(C1,C1)<>" " THEN 5000
4970 POSITION C8,C23:PRINT "PRESS RETURN FOR MORE":GET #C7,T
4980 FOR I=C23 TO C18 STEP -C1:POSITION C0,I:PRINT "{DELETE-LINE}";:NEXT I
4990 GOTO 4910
5000 CLOSE #C1
5010 POSITION C6,C23:PRINT "PRESS RETURN TO CONTINUE";
5020 GET #C7,T
5030 GOTO 1810
8000 OLDSTICK=C15:STICKWAIT=C0:BLINKWAIT=C0
8010 ROWSAVE=R:COLSAVE=C
8020 CH=CHAR:R=INT(CH/16):C=CH-16*R
8040 POSITION C2,C20:PRINT "MOVE CURSOR WITH JOYSTICK"
8050 PRINT "PRESS BUTTON WHEN READY"
8100 T=STICK(C0):IF T=C15 THEN OLDSTICK=T:GOTO 8200
8110 IF OLDSTICK<C15 THEN IF STICKWAIT>C0 THEN STICKWAIT=STICKWAIT-C1:GOTO 8200
8120 STICKWAIT=C1:IF OLDSTICK=C15 THEN STICKWAIT=C4
8130 OLDSTICK=T:POKE CONSOL,C0
8135 POSITION C+C2,R+C9:PRINT "{ESC}";CHR$(CH);
8140 ON T+C1 GOSUB 2200,2200,2200,2200,2200,2240,2220,2230,2200,2260,2280,2270,2200,2250,2210,2200
8150 IF R<C0 THEN R=C7
8155 IF R>C7 THEN R=C0
8160 IF C<C0 THEN C=C15
8165 IF C>C15 THEN C=C0
8170 CH=16*R+C
8180 BLINKWAIT=C0
8200 T=STRIG(C0):IF T=OLDTRIG THEN 8300
8210 OLDTRIG=T:IF T<C1 THEN 8400
8300 IF PEEK(KEYCODE)<>C255 THEN 8400
8310 IF BLINKWAIT>C0 THEN BLINKWAIT=BLINKWAIT-C1:GOTO 8100
8320 T=HOME+40*(R+C9)+C+C2
8330 POKE T,ASC(CHR$(PEEK(T)+C128))
8340 BLINKWAIT=C2:GOTO 8100
8400 POSITION C+C2,R+C9:PRINT "{ESC}";CHR$(CH);

```



```

8410 R=ROWSAVE;C=COLSAVE
8420 T=PEEK(KEYCODE):IF T<>C255 THEN CH=-C1:GOSUB 9300
8430 POSITION C4,C20:PRINT "{2 DELETE-LINE}":RETURN
9000 POSN=CHAR:IF POSN<C96 THEN POSN=POSN-32:IF POSN<C0 THEN POSN=POSN+C96
9010 POSN=POSN*C8:CURRENT=CHGEN+POSN
9020 FOR I=C0 TO C7:N=PEEK(CURRENT+I)
9030 POSITION C2,I
9040 FOR J=C0 TO C7:RVS=C0:N=N+N
9050 IF N>C255 THEN RVS=C1:N=N-C256
9060 STATE(I,J)=RVS:T=RVS+RVS+C2
9070 PRINT "{ESC}";DOT$(T,T);
9080 NEXT J:NEXT I:RETURN
9100 C$=CHR$(CHAR):POSITION C15,C0:PRINT "{ESC}";C$
9110 FOR I=C2 TO C6:POSITION I3,I
9120 FOR J=C1 TO 5:PRINT "{ESC}";C$;:NEXT J
9130 NEXT I
9140 RETURN
9200 I=LEN(FILE$):J=CHGEN-I-C1
9210 FOR T=I TO C1 STEP -C1
9220 N=ASC(FILE$(T)):POKE J+T,N
9230 IF N=ASC(";") THEN RETURN
9240 NEXT T:RETURN
9300 T=PEEK(KEYCODE)
9310 IF T>=C128 THEN T=T-C128:REM ZAP CTRL
9320 IF T>=C64 THEN T=T-C64:REM ZAP SHIFT
9330 IF T=60 THEN POKE LOCK,PEEK(KEYCODE)-T:REM xxx-LOCK
9340 IF T=39 OR T=60 THEN POKE KEYCODE,28:REM RVS OR LOCK{RIGHT}ESC
9350 GET #C7,T:IF T>=C128 THEN T=T-C128
9360 RETURN
9900 PRINT "{CLEAR}":POKE CRSINH,C0:POKE LOCK,C64
9910 END
30000 DIM CR$(1):CR$=CHR$(155):POKE 1536,PEEK(756):C0=0:C6=6
30010 GRAPHICS 2:OPEN #7,4,C0,"K":POKE 752,1
30020 SETCOLOR C0,8,12:SETCOLOR 3,9,4:SETCOLOR 2,C0,C0
30030 PRINT #C6;CR$;CR$;CR$;CR$;
30040 PRINT #C6;" {12 C}"
30050 PRINT #C6;" {C} FONTEdit {C}"
30060 PRINT #C6;" {12 C}"
30070 PRINT " COPYRIGHT (C) 1980"
30075 PRINT " THE CODE WORKS"
30080 PRINT "{DOWN} PRESS RETURN TO BEGIN.";
30090 GET #7,T:POKE 752,C0:CLR :GOTO 100

```

## How Fontedit Works

Most of the code in Fontedit is concerned with three things: reading commands from the user, maintaining the screen, and modifying the character generator.

Fontedit's use of memory deserves some comment: Fontedit grabs the highest four pages of memory to hold the new character generator. Putting it there keeps it out from underfoot of the Atari. Also, for reasons explained later, Fontedit must take another page of memory just before the generator. Since we therefore have some memory which is (largely) unused, we may as well put something useful into it. The most useful thing for the program to save there is the name of the font that is in the character generator. If Fontedit saves the name of the font, any program that needs a special font (such as the Knotwork program) can look to see if the font it needs is already there. If so, it doesn't need to load it, saving time and effort.

A note about the word **font**: In printer's terms, a font is a set of type (all the a's and e's and dollar signs and so forth) of one particular style (e.g. Times Italic) in a particular size (e.g. 12 points, or 1/6 inch). With the rise of computer-driven typesetting, **font** is coming to mean simply a set of characters all in a certain style. Since the computer can change the size of the type at the operator's whim, type size is much less important. (With metal type, each size of type has to be made separately. With a computer, the size is changed electronically, and there is just one shape for all sizes.) Also, the computer sets type photographically, so there are no actual metal type slugs being dealt with. We are using **font** with the new meaning of 'character style', since it is convenient and meaningful.

### === Arrays ===

- BIT()** Each element of **BIT()** contains a power of two, from 1 to 128. It is used to turn dots on and off in the character generator. (To turn a dot on, we add the appropriate power of two. To turn it off, we subtract.)
- PRISTINE()** Holds a copy of the character pattern being edited, but from a time before any changes were made to it. If we need to Undo those changes, we just copy the character pattern back from **PRISTINE()**. **PRISTINE()** is changed only by the Edit and Load commands.
- STATE()** Keeps track of which dots are on or off. **STATE()** is an 8x8 array corresponding to the 8x8 square in the screen, with one item per dot. Each item can hold either a 1, for a white dot, or a 0, for a black dot.

### === Variables ===

- BLINKWAIT** Tells how long we must wait before blinking the cursor again.
- C** Tells which column in the dot pattern the cursor is in.
- CHAR** Holds the ATASCII code for the character being edited.
- CHGEN** Holds the address of our character generator in memory. Also, briefly, while setting up, it holds the address of the actual end of memory (as opposed to where the Atari thinks it ends).
- CMD** Holds the ATASCII code for the command just entered from the keyboard.
- CMD\$** Holds the character for the command just entered from the keyboard.
- CURRENT** Tells where in memory the current character's dot pattern begins.
- DOT\$** Holds the characters needed to blink the cursor under both white and black dots.
- EOM** Holds the number of the first page beyond the end of real memory. (A page is a 256-byte chunk of memory. There are several circumstances under which the 6502 can easily use only 256 bytes of memory around a certain spot. Also, there are times when it is more convenient to figure memory size in chunks bigger than a byte.)
- FILE\$** Holds the name of the file being opened for loading or saving a font. It is also used as temporary storage in a few places.
- HASDOS** Notes whether or not the Atari knows how to talk to disks. If it doesn't, Load and Save assume cassette. Also, Font List won't work at all.
- HOME** Says where the top of the screen is in memory.
- I, J** Used all over the place in FOR loops and such.
- N** Holds numbers temporarily, when they don't need a variable of their own.
- OLDSTICK** Tells what position the joystick was in the last time we looked at it. That way we know if it has changed position since then.
- OLDTRIG** Like **OLDSTICK**, but tells if the joystick trigger (button) was pressed or not.
- ONOFF** Tells whether the cursor is on or off right now. Used to change it properly when the time comes to blink the cursor.

**POSN** Tells how far from the start of the character generator the current character's pattern starts.  
**R** Tells which row in the dot pattern the cursor is on.  
**RVS** Tells if the dot the cursor is on is white (1) or black (0).  
**STD** While we're setting things up, STD says if we need to copy the standard character generator into our generator.  
**STICKWAIT** Tells how long we have to wait before responding to the joystick again. (We can respond to the joystick faster than the user can realize we've responded, so we have to slow down to his speed.)  
**T** Like N; it holds a lot of things which are of just passing interest.  
**=== Constants ===**  
**ABORT** For a Font List, if the user presses the key CHR\$(ABORT), the listing is stopped.  
**ATTRACT** Points to the Attract Mode timer in memory. Every so often, we stuff a zero in there to prevent Attract Mode from being turned on.  
**Cnnn** Any variable which is a C followed by a number is that number. (For example, C1 is the same as just 1.) Those particular numbers are stored in variables because they are used often enough that it takes less room to keep them in a variable than it does to use them as numbers all over.  
**CHBAS** Tells where in memory the Atari keeps the address of its character generator. In the course of setting up, we have to tell the Atari that our generator is the one to use. We tell it by changing the address stored here.  
**CRSINH** Points to where the Atari keeps track of whether to display a cursor or not. We tell it not to, as soon as possible.  
**ERRSAV** Points to where the Atari stows the error number after a error has been TRAPped.  
**HATABS** Points to the Atari's list of known devices.  
**KEYCODE** Points to where the Atari stores the keycode of a pressed key.  
**LMARG** Points to where the Atari keeps the size of the left margin of the screen.  
**LOCK** Points to where the Atari keeps track of whether or not the {CAPS} key has been pressed.  
**RAMTOP** Points to where the Atari keeps its idea of the size, in pages, of memory.  
**TOPSCR** Points to where the Atari remembers where screen memory starts.

**===== The program =====**

100-120

Set up names for frequently used numbers. Storing the numbers in variables, instead of using them directly, makes the program smaller.

130 Give names to interesting locations in memory. The uses of the various locations are explained elsewhere.

- 140 Choose which character will abort a Font List. Set the left margin of the screen to be two columns wide.
- 200-210  
Allocate the arrays and strings.
- 300 DOT\$ is used in printing the big picture of the character currently being edited. It contains dark and light squares, and dark and light squares with little dots in them. The cursor blinks by alternating between squares with and without little dots. (The dot was originally the {CLEAR} character, but Fontedit changes it into the little dot while starting up.) CMD\$ holds a list of all the known commands.
- 310 Set up BIT() to contain the powers of two. BIT() is used to turn on and off single bits in memory.
- 320 Assume that this is a cassette-only system. We'll try to disprove that assumption.
- 330-350  
Search the Atari's known-devices list to see if there is a device named D: in it. If there is, we'll assume it's a disk, and that this particular Atari has disks attached.
- Find (or make) room for a new character generator**
- (CHGEN plays a dual role in here: it first points to the actual end of memory, and later points to where the character generator will live.)
- 1000 Make the Atari use its built-in character generator for a while. Find out where the Atari thinks memory ends.
- 1010-1020  
See if that is really where memory ends. The check is quite simple: we find out what's there, and put something else in its place. If what we put there stays, there is writable memory there. In that case, we put back what we took out, and try again, a little higher up (one page, to be exact).
- 1030 EOM and CHGEN now point to the actual end of memory. Put the new character generator four pages back from the end of memory.
- 1040 Check if there's already a character generator there. If there is, some work is saved, since we don't have to put one there. (If STD is non-zero, we must copy in the standard one.) Location 1536 was set up by the framework (at line 30000) to be the address of the old character generator. (1536 is the first byte of page 6 of memory. Atari swears they'll never use page 6.)
- 1060 We now reserve the memory we need for our new generator. An apparently unneeded page of memory is also grabbed. In fact, the page is needed to protect the generator against {CLEAR}s. The people who wrote the code that clears the screen assumed that there would never be anything in memory after the screen memory. On that assumption, they allowed a {CLEAR} to erase memory after the end of the screen, as well as the screen itself. The seemingly wasted page of memory gives them someplace harmless to clear. (Also, we can use some of that page ourselves, as you will see later.) Next, we issue a GRAPHICS 0, to force the screen memory out of our character generator. Finally, we make the screen black again.
- 1070 If there's already a character generator available, make it the one the Atari uses, and skip copying in the standard one.



1080 Copy the Atari's built-in character generator to the place where we're keeping ours. Taken step-by-step: The string of "garbage" is actually a small piece of machine language. The machine code copies 1024 bytes (four 'pages') of memory from one place to another, very rapidly. The ADR() function tells us where the string (and therefore the machine code) is in memory, so we can use it. The USR() function runs the machine code, and tells it to copy from the Atari's generator (224\*C256) into ours (CHGEN). The machine code itself follows:

```

FR0    =    212        ;Handy page zero space
FROM    =    FR0        ;Where to copy from
TO      =    FR0+2      ;Where to copy to
;
;      As this code will run anywhere, no
;          starting address will be given.
;
      PLA                ;Throw away number of arguments to USR()
      PLA                ;Get high byte of FROM address
      STA FROM+1         ; and save it
      PLA                ;Low byte of FROM address
      STA FROM
      PLA                ;High byte of TO address
      STA TO+1
      PLA                ;Low byte of TO address
      STA TO
;
      LDX #4            ;Copy four pages of memory
;
PAGE    LDY #0          ;Set up to copy 256 bytes
;
BYTE    LDA (FROM),Y    ;Grab a byte
      STA (TO),Y        ; and stow it where it goes
      INY                ;Advance to next byte
      BNE BYTE          ; (assuming there is one)
;
      INC FROM+1        ;Page copied. Advance to next page.
      INC TO+1          ; (Advance both FROM and TO, together.)
      DEX                ;Knock another page off the count
      BNE PAGE          ;If any pages are left, copy 'em
;
      RTS                ;All done. Go away.

```

1090 Change the {CLEAR} character to the small dot we use in the big picture.

1100 Note that the font in memory has no name.

1150 Make the Atari use our (possibly brand-new) character generator.

1160 Turn off the Atari's cursor (it's ugly, and we're going to supply our own).

1170 Print the 8x8 square of dots in preparation for the big picture of the character being edited.

1200-1210

Print the entire character set on the screen. The {ESC} is printed to tame things like {UP} and {CLEAR} (which would otherwise do nasty things to the display).

1250-1280

Put the command menu on the screen. Read a character and its description from the DATA. If there was no character, we're done. Otherwise, print it, and go back for more.

1300-1340

The data to be printed as the menu.

1400 Data that describes the small dot used in the big character picture.

**The character generator has been set up.  
It's time for other sundry preparations.**

1500 The joystick was last seen straight up. The joystick trigger (button) was last unpressed. There is be no delay in responding to the joystick.

1510 Put the cursor at row 0, column 0 of the character dot square. Make the cursor dot be on. (It will go off as soon as we hit the cursor-blinking code. We do this seemingly backwards because ONOFF says which way the cursor is now. To force the cursor the way we want it, we pretend that the cursor is the other way right now, and let the cursor-blinking code straighten things out.)

1520 The character being edited is the at-sign. (It's gotta be something, and that's as good as anything.) Find the top of the screen in memory. We'll need to know for the Copy and Edit commands.

1530 Print a few at-signs, and copy the dot pattern into the big square, so that it's obvious where the dots are.

1540 Remember the current dot arrangement, so it can be restored (after changing) if the user changes his mind.

1550 Skip the time-delay code.

**Come here after an error message is printed.**

1800 Kill some time, to let the user read the message.

1810-1820

Clear out the bottom of the screen, erasing the message. It is cleared from the bottom upwards. Were it from the top-down, the user would see the text scroll up as the lines were deleted. That looks rather odd, so we go from bottom to top.

1900 Make the cursor blink immediately.

**===== Here starts the main loop of the program. =====**

2000 Find out whether the dot at the cursor is on (RVS=1), or off (RVS=0).

2010 Look at the joystick, and see how it's being pushed. If it's straight up (T=15), remember that it was straight up, and go look at the button.

2020 If the stick wasn't upright the last time we looked, wait a little while. If we should wait longer still, go look at the button instead of responding to the joystick.

2030 We've waited long enough. It's time to respond to the joystick. Remember the new position of the joystick, and arrange for another wait next time around.

2040 Restore the dot where the cursor is to what it was before we put the cursor there.

2050 Make the built-in speaker click, to give a little audible feedback.

- 2100 The joystick can be in any of eight positions. Go off and do whatever's right for the position it's in. Come back here when we're done.
- 2110 If the cursor went off the left edge of the square, put it at the right edge.
- 2120 Similarly, if it went off the right edge, put it at the left edge.
- 2130 If it went off the top, put it on the bottom.
- 2140 If it went off the bottom, put it on the top.
- 2150 Make the cursor seem to be on, so the cursor-blinking code will turn it off. Arrange for the next blink to be immediately.
- 2170 Since the user is clearly still there (he's playing with the joystick, after all), turn off Attract Mode so that the screen will stay black. Go look at the button.

**Subroutines to move the cursor, one per joystick position.**

- 2200 Joystick is upright or impossible. Do nothing.
  - 2210 Joystick is pushed up. Move the cursor up one row.
  - 2220 Joystick is up and right. Move the cursor up and right one place.
  - 2230 Joystick is pushed right. Move the cursor right one column.
  - 2240 Joystick is down and right. Move the cursor down and right.
  - 2250 Joystick is pushed down. Move the cursor down one row.
  - 2260 Joystick is down and left. Move the cursor down and left.
  - 2270 Joystick is pushed left. Move the cursor left one column.
  - 2280 Joystick is up and left. Move the cursor up and left.
- (All of this could have been done with arrays which were indexed by the joystick position code. In this case, I decided to use a bunch of subroutines instead, on the vague impression that the subroutines together take up less room than the arrays and the code to fill them.)
- 2300 Look at the joystick button (or trigger). If it's in the same position as it was last time we looked, ignore it.
  - 2310 Remember its new position. If it's not pressed now, ignore it.
  - 2320 It's time to repaint a dot. Change the dot under the cursor to the opposite of what it is now (and remember it in RVS).
  - 2330 If the dot was turned off, turn off the appropriate bit in the character generator.
  - 2340 If the dot was turned on, turn on the appropriate bit.
  - 2350 Print a new dot on the screen, in the proper place.
  - 2400 See if a key was pressed while we weren't looking. If one was, go off and do something with it.
  - 2410 Is it time to blink the cursor yet? If not, go off and start things over by looking at the joystick again.
  - 2420 Turn the cursor off if it was on, and vice versa. Find out if the dot under the cursor is light or dark.

2430 Select the proper character to print at the cursor location, depending on whether the dot is on or off, and whether the cursor dot is to be there or not. Print it at the right place.

2440 Wait a while before the cursor blinks again. Go look at the joystick again.

**A key was pressed. Do what it says to do.**

3000 Put the dot under the cursor back to normal (make sure the dot is visible).

3010 Go off and find out which key was pressed.

3050 If it was a lower case letter, make it upper case.

3060-3100

Search CMD\$ for the pressed key. If it's not there, ignore it, and go look at the joystick. If it is there, go off and do the appropriate thing.

**{CLEAR} - Clear the current character to blank**

3200 Store zeroes into all eight dot rows of the character.

3210 Redraw the big picture of the character, then go look for something else to do.

**R - Reverse the dots (switch light and dark) of the current character**

3300 In the generator, wherever there is a 1 bit in the character picture, put a zero there, and vice versa.

3310 Redraw the big picture, and start the main loop over.

**E - Select a new character to edit**

3400 Go off and let him select a character to edit. If he changed his mind (CH<0), don't do anything.

3420 Note the new character being edited.

3430 Put the usual set of standard-sized characters on the screen (one alone, and 25 in a square). Put a big picture of the character in the 8x8 square of dots.

3440 Remember the starting dot arrangement, so the user can get it back if he wants it.

3450 Put the cursor in the upper-left corner of the square, and start the main loop over.

**U - Undo any changes the user's made to the character**

3500 Put the saved copy of the original arrangement back into the character generator.

3510 Redraw the big picture of the character, and restart the main loop.

**{UP} - Roll the dots making the character up one row**

3600 Save the top row of the character against destruction.

3610 Move each of the other seven rows up one row.

3620 Put the saved top row back in as the new bottom row.

3630 Redraw the big picture of the character, and back to the main loop.

**{DOWN} - Roll the dots of the character down one row.**

3700 Save the current bottom row for later.

3710 Move each row down to the one below it. Note that the loop to do this runs backwards, so to speak. If it went from 0 to 6, we would copy row 1 into row 2, and then row 2 (which is now the same as row 1) into row 3, then row 3 (which is now the same as row 2, which is the same as row 1) into row 4, and so on. We would wind up with all the rows identical, rather than just moved. Thus, we copy row 6 to row 7, then row 5 to row 6, and so on.

3720 Put the old bottom row in as the new top row.

3730 Redraw the big picture, and restart the main loop.

**{LEFT} - Roll the dots of the character one column left**

3800-3830

For each row of dots in the character, grab the row into T. Move all the dots left by one dot, putting a black spot where the rightmost dot used to be. If a dot was pushed off the left end, push it around to the right end, to where the black spot was put before. Put the shifted row of dots back into where it came from. Do the same thing to the other seven rows.

3840 Redraw the big picture, and go back to the main loop.

**{RIGHT} - Roll the dots in the character right one place.**

3900-3930

For each row of dots in the character, grab the row into T. Move the dots right one place (by dividing by 2). If a dot went off the right end, (the old row was an odd number, so the quotient has a fraction, and we must...) move the dot around to the left end of the row. Put the modified row back in place of the old one. Do this for each of the other seven rows.

3940 Redraw the big picture, and restart the main loop.

**C - Copy the dot pattern of another character into this one.**

4000 Go off and let the user select a character to copy. If he changed his mind (CHK0), don't do anything.

4030 Find the character's position in the character generator. (This code is determined solely by the way Atari arranged the characters in the generator. If I'd had my way, the ATASCII code would have sufficed.)

4040 Convert the position in the generator to a position in memory.

4050 Copy the dot pattern into the character being edited.

4060 Redraw the big picture, and go back to the main loop.

**A - Restore Atari's dot pattern for the character**

4100 Find out where the character's pattern is in the built-in generator.

4110 Copy the built-in pattern to the character being edited.

4120 Redraw the big picture, and return to the main loop.

**S - Save the current font in a file**

4200 Ask which file to save the font in. If he just presses {RETURN}, he changed his mind, so just go back to the main loop.

4210 If anything goes wrong while we're saving, go to line 4300

4220 Open the file for writing.

4230 Say what we're doing.

4250-4260

Write out all of the character generator to the file. While that's happening, move a small white line across the character array on the screen. The line shows how much of the task has been done, and that we're doing something useful, and haven't died. The line is made by switching the white and black dots in the byte that's being written to disk. (While the blank character's pattern is being written, the line is suppressed. Were it not, the line would flash all over the screen, which is somewhat disconcerting.) When we're done with all that, say so.

4270 Close the file, and change the name of the loaded font. Kill some time to let the user read the "SAVE COMPLETE" message. Erase the message, and return to the main loop.

4300-4320

No luck saving the font. Say that it's impossible, close the file (in case we died in the middle of saving it), change the name of the in-memory font (despite the save not working), and return to the main loop.

#### **L - Load a font from a file**

4400 Ask the user what file to get the font from. If he presses {RETURN}, he'd rather keep the current one, so leave it alone.

4410 If anything goes wrong while loading, go to line 4500

4420 Open (or at least attempt to open) his file.

4430 Say what we're doing.

4450-4460

Read in the new font, and stuff it away in the character generator. All the while we're loading the new font, we move a little line across the character array on the screen. It's the same line as for Save, for the same purpose, and it's done the same way. When we're done, redraw the big picture of the current character (since it may have changed on us when the new font was loaded.) Also, print a message saying that the task is done.

4470 Close the file, and store the name of the newly-loaded font into where we keep font names. Give the user some time to notice the "LOAD COMPLETE" message. Erase the message, and go back to the main loop.

4500-4540

Say that we couldn't load the font completely. (Unfortunately, we can't restore the font to what it was before the abortive load.) Close the file, and say that there is no font in memory (since we were so rudely interrupted). We do that by making one of the characters in the font name (which is stored just before the generator) be one that cannot occur in a real font name, by POKEing a zero byte (or a null) into the name. The Atari doesn't allow nulls in file names, so they can't be part of font names, either.

4600-4670

Here we are going to get a file name, and make sure it has everything the Atari likes file names to have.

4600 "Press" the {CAPS} button, so that the file name will be typed in using all caps. Save the old position of the caps-lock, so we can restore it later.

4610 Print the proper question, as directed from above, and get a reply.

4620 Erase the question, and the reply. Restore the caps-lock to what it was before we meddled with it.



4630 If he just pressed {RETURN}, we have nothing to do. Do just that.

4640-4660

See if there's a colon (and therefore a device) in the file name. If there is, we don't need to assume a device.

4670 No colon in the file name, so there wasn't a device. Assume the disk, unless there is no disk, in which case assume the cassette.

4700-4720

See if there's a period (and therefore an extender) in the file name. If there is, all is well, and we're done.

4730 No extender. Tack on an extender of ".FNT" (for FoNT) to keep font file names separate from others.

4740 All done. Hand the file name back to whoever it was that wanted it.

#### **F - List the fonts saved on a disk**

4800 See if there's a disk to lists the fonts on. If not, say so, and give up.

4810 Ask which drive we should look on, get the reply, and erase the whole mess. If he pressed {RETURN}, he probably changed his mind (although he may have assumed that {RETURN} gave him drive 1, a plausible, but wrong, assumption). That being the case, go back to the main loop.

4820 Check the drive number he entered to see if it's legal. If not, complain and do nothing.

4830 Set up to read the directory of that drive.

4840-4850

Attempt to open the directory for reading. If it fails, say so, and forget he ever wanted anything.

4900 Read a line from the directory. (Incidentally, the directory is given to the program in exactly the same format as you see it using the DOS.) We read a line before we enter the font-name-printing loop so that we are always a little ahead of the printing. That way, we can tell if we've run out of names before we have to decide which "PRESS RETURN" prompt to print.

4910-4960

Fill lines 18 through 21 on the screen with four font names each. If we run out of names before we run out of screen, go directly to the "CONTINUE" message. Again, we read a name AFTER we print one so we will know that we've run out of names before it's too late.

4965 The screen is full. Was it filled by the last font name in the directory? If so, go off and print the "CONTINUE" message. It is for this question that we read names the way we do. Suppose we read the names just before we printed them (by doing INPUT #1,C\$:PRINT C\$(3,10)), instead of keeping one ahead as we do. Also suppose that there are just enough fonts on the disk to fill the screen once (sixteen, as things are set up). When the screen is full, we will have read all sixteen names from the directory, and there will be none left on the disk, so we should print "PRESS RETURN TO CONTINUE" instead of "PRESS RETURN FOR MORE" (since there are no more names). However, since we won't yet know what the next thing we will read is, we must assume it's another name. The user is thus told that the names were all printed last screenful, after having been told that more names were coming. This unfortunate occurrence doesn't happen with Fontedit, because Fontedit always knows what comes after the last name it printed on the screen, since it went and looked. It therefore knows what it should do when the screen fills up.

- 4970 Say that there are more fonts lying in wait for him, and kill time till he presses {RETURN} (or any other key, for that matter; why be picky?).
- 4980 Erase all the names we just finished printing. As with erasing error messages, we do it bottom to top to avoid unsightly scrolling.
- 4990 Go off and print another screenful.
- 5000 Close the directory; we're done with it.
- 5010-5030  
Say we're done, and wait for him to finish perusing the font names. When he says he's done, go back and let him do something else.
- Select a character to Edit or Copy, with the joystick.**
- 8000 Assume the joystick is straight up. Make an instant response when it moves, and make the cursor blink immediately.
- 8010 Remember where the pattern cursor is, so we can put it back in the right place when we're done.
- 8020 Start with the cursor on the character being edited.
- 8040-8050  
Say what the user should do to select a character.
- 8100 Peer at the joystick. If it's straight up, remember the fact, but don't do anything about it.
- 8110 If the joystick wasn't straight up last time, either, wait a bit before responding to it. This slows us down to the user's speed.
- 8120 Choose how long to wait before the next response to the joystick. (Thus, the cursor starts slow, but moves fast once it gets started.)
- 8130 Remember what position the joystick is in now. Also, make the built-in speaker click as audible feedback.
- 8135 Put the character under the cursor back to normal. (Make sure it's normal video, and not reversed.)
- 8140 Move the cursor according to how the joystick's been pushed.
- 8150-8165  
Keep the cursor within limits. Note that the limits are different from those on the pattern cursor.
- 8170 Figure out which character the cursor is on.
- 8180 Make the cursor appear immediately.
- 8200 Look at the joystick button. If its position is unchanged, ignore it.
- 8210 The position has changed. Make a note of the new position. If it's pressed now, the user's chosen a character, so hand it back.
- 8300 Look at the keyboard. If he pressed something, he doesn't want to select a character after all, so hand nothing back.
- 8310 Nothing to do, so try to blink the cursor. If it isn't yet time to blink it, go back and look at the joystick again.
- 8320 Time to blink it. Figure out where the cursor is in screen memory.

8330 Change it from normal to reverse video, or vice versa. The effect of the ASC(CHR\$(...)) is to throw away any part of the number greater than 255; in other words, ASC(CHR\$(...)) returns the remainder of .../256.

The reason we do a POKE to screen memory, instead of the usual POSITION followed by PRINT, is that we need to be able to display a reverse video {ESC}. If you try to print one, you'll get a carriage return, which is invisible. By changing the screen with POKE, we bypass the Atari's printing routines, and can get whatever we want on the screen. Were it not for that one single thing, POSITION and PRINT would be used here, too.

8340 Schedule the next cursor blink, and go off to look at the joystick.

8400 A character's been chosen! Change the character back to normal video.

8410 Restore the pattern cursor again.

8420 If a key was pressed, return an invalid character, and go grab the key, so that we get a keyclick.

8430 Erase the directions for character selection, and hand the character back.

### **Print the big picture of the character being edited**

9000-9010

Figure out where the character is in the character generator. As with the Copy command, the formula is used solely because it works. There's no arcane theory behind it.

9020-9080

Grab one row of dots for the character from the character generator. Put the cursor at the start of the proper line in the 8x8 square. To fill in the dots in the row, we must do the following: For each column, shift the dot row left by one (by adding it to itself). If a dot falls off the left end (if the result is greater than 255), print a white spot in the column. Otherwise, print a black spot. In either case, remember which kind of dot it was, so we can diddle it later. When we've done all the rows and columns, we're done, so return to whoever called us.

### **Print the sample characters beside the big picture**

9100 Print the character being edited all by itself beside the big picture.

9110-9130

Below it, print five rows of five of the character being edited, so that the user can see how they look all run together.

9140 All done, so return.

### **Store the font name just in front of the character generator**

9200 Find out how long the name is. Figure out where we have to put it so that it will end just before the generator starts, so that name and generator touch.

9210-9240

Store the font name right up against the generator, storing the last letter first, the next-to-last letter next, and so on. Stop after storing the colon for the device. (We store the colon so that later, if we want, say, "FIN.FNT", we won't be fooled if "ELFIN.FNT" is loaded, since ":FIN.FNT" can't be confused with ":ELFIN.FNT".) When the name is saved, return to where we came from.

## Get a single keypress, even if it's {RVS} or {CAPS}

- 9300 Find out the keycode (not the ATASCII code) for the pressed key.
- 9310 If the {CTRL} key was pressed with it, pretend it wasn't.
- 9320 Likewise, if {SHIFT} was pressed with the key, pretend it wasn't.
- 9330 If the key pressed was the {CAPS} (or {LOWR}, same thing) key, change the Atari's caps-lock setting for it. (We don't intend that the Atari ever see that {CAPS} was pressed.)
- 9340 If the key pressed was either {CAPS} (whose keycode is 60) or {RVS} (with the Atari logo on it; its keycode is 39), pretend that it was really the {ESC} key (keycode 28) that was pressed. This is because {CAPS} or {RVS} alone isn't enough to make a GET statement return a value. The GET will sit there waiting for some other key to be pressed before letting the program continue. That doesn't suit our plans, so we make GET think that {ESC} was pressed. (We could just skip the GET completely for {CAPS} and {RVS}, but we want a key-click to be heard, and we have to do a GET to get one. This is a subtle design point: by making the key-click, we tell people that the key-press had its effect, even though nothing visible happens.)
- 9350 Get the ATASCII code for the pressed key. (We could have worked with the keycode, as we did for {RVS} and {CAPS}, but that isn't nearly as clear to the reader. For {RVS} and {CAPS}, the keycode method is forced on us, since they have no ATASCII code at all.) If it is reverse video, make it normal.
- 9360 Now that we have the key, hand it back.

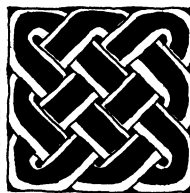
## Q - Quit the program

- 9900 Clear the screen, turn the Atari's cursor back on, make sure that the {CAPS} button has been "pressed" (since the Atari doesn't like lower case commands).
- 9910 Free at last!

---

# Knotwork

Knotwork (or, more accurately, Celtic interlace) is a technique of manuscript illumination used by the 7th and 8th century Irish monks. In essence, a piece of knotwork is a drawing of interlaced cords. (The example to the right is plain knotwork.) Were it left at that, it wouldn't be very interesting. However, unadorned knotwork is never found. What are called **breaks** are always added. A break is basically a rerouting of the cords, so that they don't cross. By putting breaks at judiciously chosen places, quite interesting patterns can result. (The example to the left has a couple of breaks in it. Those alone are enough to make it noticeable different from the first example.) The Knotwork program lets you try your hand at constructing knotwork patterns. Or, rather, with constructing patterns within a rectangle. The Irish monks also drew knotwork in fancy shapes, and did it with things other than plain cords. Unfortunately, those approaches are well beyond the capabilities of the program. However, we hope the program will encourage some to learn more about knotwork in particular, and calligraphy in general.



The Knotwork program uses a special font (or character set) to do its knotwork with. When Knotwork starts up, it has to make sure that its font is loaded (either by loading it, or finding out that the font is already loaded). It will then sit there for a bit, setting up things, and then draw an unadorned knotwork pattern on the screen. It is up to you to put in breaks where you want them.

It's important to remember that Knotwork must have its special font in memory to do anything more than print its name. If you're using cassette, you'll want to keep a copy of the Knotwork font as the first file on a tape all its own. With disks, things are simpler. You need only keep the Knotwork font (saved as "KNOTWORK.FNT") on the same disk as the Knotwork program. (To keep the program simpler, Knotwork assumes that its font is on drive 1, and will sulk if it isn't.)

To put in breaks, you have to move the arrow (using the joystick) so that it points to where you want the break. To put in the break in, press the joystick button. (If you change your mind, push the button again, and Knotwork will take the break out.)

Due to the arrangement of places that breaks can be put, the arrow's movement is not completely straightforward. It alternates between pointing down and pointing sideways. When pointing down, it points to the right-hand end of the (potential) break; When pointing sideways, to the top of the break. Getting it positioned correctly quickly is something that comes with practice. (Not a lot of practice, fortunately.)

Sooner or later, you'll come up with a knotwork pattern that you want to save and treasure. To do so, press **S** (for Save). The Knotwork program will ask what name you want to save the pattern as. Type in a name that is from one to eight letters long, and press {RETURN}. You'll then see a phantom spot moving across the pattern, copying it into the file. When the spot has crossed the entire pattern, the pattern has been saved. The arrow will reappear, and you can continue doing knotwork. (If you don't have a disk, Knotwork will merely ask if your cassette drive is ready. No name will be needed, nor asked for. As always with cassette, after you hear the two buzzes, you'll have to press PLAY and RECORD on the recorder, and then press {RETURN} to let the Atari proceed.)

Later, when your friends are around, you can reload the pattern by pressing **L** (for Load). When asked for a name, type in the same name you gave when saving the pattern, and then press {RETURN}. The same phantom spot will reappear, putting the saved pattern back up again. As before, when it's done the whole pattern, the arrow will reappear, and you can admire it or change it, as you please. (As with Save, a name is asked for only if you have a disk. Otherwise, Knotwork just asks if the cassette drive is ready, and waits for a {RETURN} after PLAY has been pressed.)

If you want to start over without any breaks set in the pattern, press the {CLEAR} key. Whatever pattern you have drawn will be erased (by the same phantom spot as for Load and Save).

Finally, when you tire of creating knotwork patterns, press **Q** (for Quit), and Knotwork will release you to other tasks.

=====

For those of you who become interested, we are including an article "Interlacing without Erasing," which contains directions on how to do knotwork yourself, without aid of the computer. The design of the program, in fact, was inspired by the techniques described in that article. (The secret of doing knotwork was lost for many centuries, along with most of the rest of the art of calligraphy. It was

rediscovered in the mid-twentieth century by studying old manuscripts which hadn't been completed, or in which mistakes had been made.) The directions are provided courtesy of Melinda Sherbring and the Society for Creative Anachronism, Inc. (the SCA).

The SCA is a non-profit educational organization that fosters study of medieval life and culture by recreating it. The SCA is a learn-by-doing group that encourages members to adopt a persona of the period. This includes devising a unique name and coat-of-arms; attending events dressed in garb of one's chosen time, place, and profession; and behaving in an honorable and chivalrous manner. The interlacing article is an outgrowth of an SCA-sponsored workshop on calligraphy and illumination. It was written by Mistress Eowyn Amberdrake, the Magistra Scriptorium of Caid (head of the College of Scribes in Southern California). Mistress Amberdrake is known to those outside the SCA as Melinda Sherbring, a software engineer for TRW, Inc. in Redondo Beach, Ca. For further information on the SCA, contact the Office of the Registry, P.O. Box 594, Concord, Ca. 94522.

## Knotwork Listing

```

0 REM KNOTWORK
1 REM COPYRIGHT(C) 1980 THE CODE WORKS
2 REM BOX 550, GOLETA, CA. 93017
3 REM ALL RIGHTS RESERVED
10 REM
90 GOSUB 30000
100 C0=0:C1=1:C2=2:C4=4:C6=6:C15=15:C16=16:C64=64
110 HORIZ=C1:VERT=C0:SCREENWIDTH=40
120 HEIGHT=22:WIDTH=36
140 GOSUB 4900
200 BREAKSSIZE=(HEIGHT+C2)*SCREENWIDTH
210 DIM BITS(C4,C2),ROWCHANGE(C15),COLCHANGE(C15)
220 DIM BREAKS$(BREAKSSIZE),CHSET$(128),CODE$(60),MISC$(20),FILES$(20)
230 BREAKS=ADR(BREAKS$)
300 P=ADR(CODE$)
310 BITXOR=P:GOSUB 9100
320 BITAND=P:GOSUB 9100
330 BITOR=P:GOSUB 9100
340 DATA 68,68,85,D5,68,85,D4,68,45
350 DATA D5,85,D5,68,45,D4,85,D4,60,=
360 DATA 68,68,85,D5,68,85,D4,68,25
370 DATA D5,85,D5,68,25,D4,85,D4,60,=
380 DATA 68,68,85,D5,68,85,D4,68,05
390 DATA D5,85,D5,68,05,D4,85,D4,60,=
410 FOR I=C0 TO C15:READ R:ROWCHANGE(I)=R:NEXT I
420 FOR I=C0 TO C15:READ C:COLCHANGE(I)=C:NEXT I
430 DATA 0,0,0,0, 0,1,-1,0, 0,1,-1,0, 0,1,-1,0
440 DATA 0,0,0,0, 0,1,1,1, 0,-1,-1,-1, 0,0,0,0
800 CHSET$(C1)="dcbaihihmmkknnnn"
805 CHSET$(17)="qqqrrrrruuuuxxxx"
810 CHSET$(33)="pooottssswvvvyxxx"
815 CHSET$(49)="gefajhjhllkknnnn"
820 CHSET$(65)=" {4 D} {4 D}"
825 CHSET$(81)=" {4 C} {4 C}"
830 CHSET$(97)=" {8 B}"
835 CHSET$(113)=" {8 A}"
850 FOR I=C0 TO C1:FOR J=C0 TO 3
860 READ T:BITS(J,I)=T
870 NEXT J:NEXT I
880 DATA 1,4,1,4, 2,2,8,8
900 GRAPHICS C0:POKE 752,C1:POKE 82,C0:POKE 756,PEEK(106)+C1
910 SETCOLOR C2,C0,C0
1000 POSITION C0,C0:PRINT "ONE MOMENT, PLEASE.";
1100 FOR I=C0 TO BREAKSSIZE-C1 STEP 80
1110 FOR J=C1 TO 39 STEP C2
1120 BREAKS$(I+J)="{, P}"
1130 NEXT J
1140 FOR J=41 TO 79 STEP C2
1150 BREAKS$(I+J)=" 0"
1160 NEXT J
1170 PRINT ".";
1180 NEXT I

```



```

1500 T=C64+C16:S=BREAKS+SCREENWIDTH
1510 FOR I=C1 TO HEIGHT
1520 POKE S,T+C1
1530 POKE S+C1,PEEK(S+C1)+C4
1540 POKE S+WIDTH,PEEK(S+WIDTH)+C1
1550 POKE S+WIDTH+C1,T+C4
1560 T=USR(BITXOR,T,C16):S=S+SCREENWIDTH
1570 NEXT I
1600 T=96+C16:S=BREAKS+C1:N=HEIGHT*SCREENWIDTH
1610 FOR I=C1 TO WIDTH
1620 POKE S,T+C2
1630 POKE S+SCREENWIDTH,PEEK(S+SCREENWIDTH)+8
1640 POKE S+N,PEEK(S+N)+C2
1650 POKE S+N+SCREENWIDTH,T+8
1660 T=USR(BITXOR,T,C16):S=S+C1
1670 NEXT I
1700 POKE BREAKS,C64:POKE BREAKS+WIDTH+C1,C64
1710 POKE BREAKS+N+SCREENWIDTH,C64:POKE BREAKS+N+SCREENWIDTH+WIDTH+C1,C64
1900 FOR I=C0 TO HEIGHT+C1
1910 R=I*SCREENWIDTH
1920 FOR J=C0 TO WIDTH+C1
1930 POSITION J,I:PUT #C6,32:POSITION J,I
1940 PUT #C6,ASC(CHSET$(PEEK(BREAKS+R+J)+C1))
1950 NEXT J:NEXT I
2000 ROW=C1:COL=C1
2010 OLDTRIG=C1:OLDSTICK=C15
2100 R=USR(BITAND,ROW,C1):C=USR(BITAND,COL,C1)
2120 DIR=VERT:CURSOR=ASC("{RIGHT}"):IF R=C THEN CURSOR=ASC("{DOWN}"):DIR=HORIZ
2200 POSITION COL,ROW:GET #C6,UNDER
2210 POSITION COL,ROW:PUT #C6,CURSOR
2220 T=STRIG(C0):IF T<OLDTRIG THEN OLDTRIG=T:IF T=C0 THEN 4000
2230 T=STICK(C0):IF T>C15 THEN 2300
2240 OLDSTICK=T
2250 T=PEEK(764):IF T<>255 THEN 3000
2260 GOTO 2220
2300 IF T=OLDSTICK AND IDLE>C0 THEN IDLE=IDLE-C1:GOTO 2220
2310 IF T=OLDSTICK AND DELAY>C0 THEN DELAY=INT(DELAY/C2):IDLE=DELAY
2320 IF T<>OLDSTICK THEN DELAY=C4:IDLE=DELAY:OLDSTICK=T
2400 POSITION COL,ROW:PUT #C6,UNDER
2410 ROW=ROW+ROWCHANGE(T):COL=COL+COLCHANGE(T)
2420 IF ROW<C0 THEN ROW=HEIGHT
2430 IF ROW>HEIGHT THEN ROW=C0
2440 IF COL<C0 THEN COL=WIDTH
2450 IF COL>WIDTH THEN COL=C0
2460 GOTO 2100
3000 POSITION COL,ROW:PUT #C6,UNDER:POSITION C0,C0
3010 T=USR(BITAND,T,63)
3020 IF T=39 OR T=60 THEN POKE 764,28
3030 GET #7,T:IF T>=128 THEN T=T-128
3040 IF T>ASC("a") AND T<=ASC("z") THEN T=T-ASC("a")+C1
3110 IF T=ASC("{CLEAR}") THEN 1000
3120 IF T=ASC("Q") THEN 5000
3130 IF T=ASC("L") THEN 3200
3140 IF T=ASC("S") THEN 3400
3150 GOTO 2200
3200 MISC$="LOAD":GOSUB 3600:IF FILE$="" THEN 2200
3210 TRAP 3300:OPEN #C1,C4,C0,FILES
3220 FOR I=C0 TO HEIGHT+C1
3230 R=I*SCREENWIDTH
3240 FOR J=C0 TO WIDTH+C1
3250 POSITION J,I:PUT #C6,32:POSITION J,I
3260 GET #C1,T:POKE BREAKS+R+J,T:PUT #C6,ASC(CHSET$(T+C1))
3270 NEXT J:NEXT I
3280 GOTO 3310
3300 POSITION C0,C0:PRINT "CAN'T LOAD FROM ";FILES:GOTO 3390
3310 POSITION C0,C0:PRINT "DONE."
3390 CLOSE #C1:FOR T=C1 TO 200:NEXT T:GOSUB 3700:GOTO 2200
3400 MISC$="SAVE":GOSUB 3600:IF FILE$="" THEN 2200
3410 TRAP 3500:OPEN #C1,8,C0,FILES
3420 FOR I=C0 TO HEIGHT+C1
3430 R=I*SCREENWIDTH
3440 FOR J=C0 TO WIDTH+C1
3450 POSITION J,I:PUT #C6,32:POSITION J,I
3460 T=PEEK(BREAKS+R+J):PUT #C1,T:PUT #C6,ASC(CHSET$(T+C1))
3470 NEXT J:NEXT I
3480 GOTO 3310
3500 POSITION C0,C0:PRINT "CAN'T SAVE INTO ";FILES:GOTO 3390
3600 POSITION C0,C0:IF HASDOS=C0 THEN 3800

```

```

3605 T=PEEK(702):POKE 702,C64:PRINT "FILE TO ";MISC$;
3610 INPUT MISC$:POKE 702,T:GOSUB 3700
3620 FILES=MISC$:IF FILES="" THEN RETURN
3630 FOR T=C1 TO LEN(FILES)
3640 IF FILES(T,T)="." THEN 3660
3650 NEXT T:FILES="D":FILES(3)=MISC$
3660 FOR T=C1 TO LEN(FILES)
3670 IF FILES(T,T)="." THEN RETURN
3680 NEXT T:FILES(LEN(FILES)+C1)="."KNT"
3690 RETURN
3700 POSITION C0,C0:PRINT "
3800 PRINT "IS CASSETTE READY FOR ";MISC$;"?";
3810 GET #7,T:GOSUB 3700:IF T>=128 THEN T=T-128
3820 FILES="C":IF T=ASC("N") OR T=ASC("n") THEN FILES=""
3830 RETURN
4000 POSITION COL,ROW:PUT #C6,UNDER
4010 POKE 77,C0
4020 M=ROW:N=COL-(DIR=HORIZ)
4030 FOR I=C0 TO C1:R=I+M
4040 FOR J=C0 TO C1:C=J+N
4050 IF C<C0 OR R<C0 THEN 4100
4060 S=BREAKS+SCREENWIDTH*R+C
4070 CH=USR(BITXOR,PEEK(S),BITS(I+J,DIR)):POKE S,CH
4080 POSITION C,R:PUT #C6,ASC(CHSET$(CH+C1))
4100 NEXT J:NEXT I
4110 GOTO 2100
4900 HASDOS=C0
4910 FOR I=C0 TO 12
4920 IF PEEK(794+3*I)=ASC("D") THEN HASDOS=C1
4930 NEXT I:RETURN
5000 PRINT "{CLEAR}":POKE 752,C0:END
8999 END
9100 READ MISC$:IF MISC$="" THEN RETURN
9110 H=ASC(MISC$(C1,C1))-48:IF H>9 THEN H=H-7
9120 L=ASC(MISC$(C2,C2))-48:IF L>9 THEN L=L-7
9130 POKE P,H*C16+L:P=P+C1:GOTO 9100
30000 C0=0:C1=1:C6=6:CHBAS=756:DIM CR$(C1):CR$=CHR$(155):OLDGEN=PEEK(CHBAS):POKE 82,2
30010 GRAPHICS 2:OPEN #7,4,C0,"K":POKE 752,C1
30020 SETCOLOR C0,8,12:SETCOLOR 3,9,4:SETCOLOR 2,C0,C0
30030 PRINT #C6;CR$;CR$;CR$;CR$;
30040 PRINT #C6;" {12 C}"
30050 PRINT #C6;" {C} KNOTWORK {C}"
30060 PRINT #C6;" {12 C}"
30070 PRINT " COPYRIGHT (C) 1980"
30075 PRINT " THE CODE WORKS"
30080 PRINT "{DOWN} PRESS RETURN TO BEGIN.";
30090 GET #7,T:PRINT "{CLEAR}"
30500 P=PEEK(106):DIM F$(20):F$="D:KNOTWORK.FNT"
30510 A=P*256:C=255-PEEK(A):POKE A,C:IF PEEK(A)=C THEN POKE A,255-C:P=P+C1:GOTO 30510
30520 P=P-5:IF P<PEEK(106) THEN POKE 106,P:GRAPHICS C0:OLDGEN=0
30530 P=P+C1:A=P*256
30540 IF OLDGEN<>P THEN 30600
30550 I=LEN(F$):J=A-I-C1
30560 FOR T=I TO C1 STEP -C1
30570 C=ASC(F$(T)):IF PEEK(J+T)<>C THEN 30600
30580 IF C=ASC(":") THEN 30690
30590 NEXT T
30600 GOSUB 4900:IF HASDOS THEN 30620
30610 F$(1,1)="C":PRINT "{CLEAR}PUT TAPE WITH ";F$:PRINT "IN CASSETTE PLAYER."
30620 TRAP 30700:OPEN #C1,4,C0,F$
30630 PRINT "{CLEAR}LOADING ";F$:C=ASC("X"):POKE 752,C1
30640 FOR I=C0 TO 1023:GET #C1,J:POKE A+I,J:PRINT CHR$(C);"{LEFT}";:C=107-C:NEXT I:
CLOSE #C1:TRAP 50000
30650 I=LEN(F$):J=A-I-C1
30660 FOR T=I TO C1 STEP -C1
30670 C=ASC(F$(T)):POKE J+T,C:IF C=ASC(":") THEN 30690
30680 NEXT T
30690 POKE CHBAS,P:CLR:GOTO 100
30700 PRINT "CAN'T LOAD ";F$;": ERROR ";PEEK(195):END

```

## How Knotwork Works

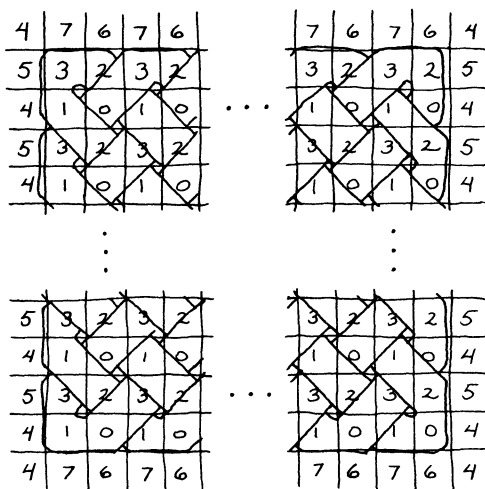
The Knotwork program is designed around a set of specially-defined characters; the characters are pieces of interlace, some with horizontal and vertical lines where breaks occur. The program uses those characters to make the knotwork pattern, rather than drawing the pattern with a high-resolution graphics mode. This makes the program simpler to write because it need only be concerned with printing the right character at the right time. It doesn't have to worry about erasing unnecessary lines, not erasing necessary ones, drawing lines that aren't there, and so forth. All the drawing and erasing considerations were taken care of when the character set used in printing was designed. Also, since a text mode is used instead of a graphics mode, the program takes less memory to run (even counting the space taken by the character generator).

The code in Knotwork has three main parts: watching the joystick to decide what should be done; keeping track of where breaks have been placed by the user; and keeping the picture on the screen accurate.

To help it in its work, Knotwork keeps a "copy" of the screen in an array. "Copy" is in quotes because it isn't really a replica; rather, it is a table which describes the breaks, with one item in the table for each character on the screen. When a break is placed (or removed), the break table is changed first. The picture on the screen is then changed according to information in the break table.

Each item in the break table has two types of information in it: some indicators as to whether there are breaks around that position, and which of eight sets of characters to choose from for that position on the screen.

To draw the knotwork correctly requires that a different set of characters be used in different places on the screen. (The sets of characters should not be confused with the character set, or font, that contains the sets of characters.) The sets are numbered from 0 to 7, and each set contains sixteen characters. There are sixteen characters in a set because each character position may or may not have a break on each of the four sides, and there are sixteen combinations of breaks and non-breaks. The drawing shows which set is used where on the screen. The main knotwork pattern draws upon sets 0, 1, 2, and 3. The top and bottom edges are printed from sets 4 and 5. The left and right edges are drawn using sets 6 and 7. The corners, which never show up as anything but blanks, are printed using set 4.

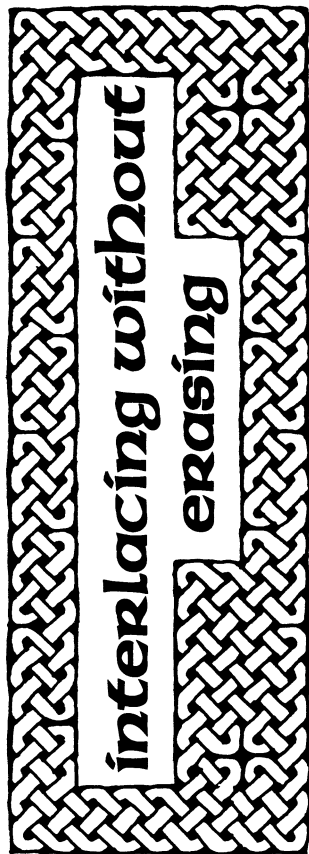


Each item in the break table contains, in the upper four bits of the item, the number of the set of characters to be used for that position of the pattern.

The break indicators are kept in the low four bits of the item. Bit 0 (=1) tells of a break at the right; bit 1 (=2) of a break underneath; bit 2 (=4) of a break to the left; and bit 3 (=8) of a break above.

The set number and break indicators are arranged so that Knotwork can easily choose, from CHSET\$, the exact character to print. It merely PEEKs at the proper item in the break table, and uses the resulting number as a subscript to CHSET\$. No fancy calculations are needed. (A nice example of arranging the data to make the program simpler.)

(Continued on page 30)



By Eowyn Amberdrake, A.A., O.L.M., O.L.

The following article is based on a class given by Mark van Stone, a calligrapher who has studied the original medieval manuscripts to learn the techniques used in their production. The insights herein presented are his.

The basis for this style of knotwork is a grid of dots, and this principle:

**Knots avoid dots; breaks join them.**

### I. The Dots

1. Put four dots at the corners of a square.
2. Put a dot in the middle of the square, and in the middle of each side of the square.
3. Put a dot in the center of each small square.
4. Construct a band of the small squares at least two squares (three dots) wide.

### II. The Knots

5. Draw two lines, just inside the borders of a diamond of dots. The ends of the lines should stop just beyond the dots.
6. Draw two lines perpendicular to the ones just drawn. The first two lines should touch the closest new line.
7. Draw another strand from the first one, with two more lines.

### V. Breaks

This one style of knotwork can get boring after a while. We can spice it up by putting breaks in the gridwork before knotting it. The rules:

A break is horizontal or vertical, ~~never~~ diagonal. A break never crosses another break.

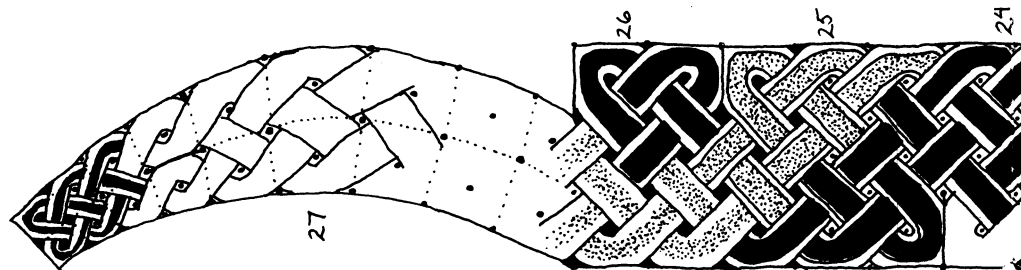
21. Draw a line between dots in from the edge.

22. Knot the strands there as if the break were an edge (and a corner).

23. A pattern of four breaks making a cross was popular with the medieval monks.

To keep your knots all in one strand, leave at least two dots between successive breaks or break patterns.

Any arrangement of breaks is acceptable, as long as it doesn't violate the rules.



8. Keep going...

9. Continue down the center of the band of squares. Draw parallel lines only when all four dots surrounding the strand appear.

10. Consider the edges. Without a full diamond of dots, the strands bounce off the edge.

11. Finish the inside of the turn by continuing the lines around their dots. The strand has just made a 90° turn.

12. To end it, draw a line through the sides of a small square, not through the centers.

13. Fill all the diamonds with strands (parallel lines).

14. Extend the untied ends to the edges, and follow the sides to the corners.

15. Finish the inner side of the turning strands. The strands have just made a 180° turn.

### III. The Corners

16. A band two squares wide needs a 4 square area for its corner. The inner edge must pivot around a square's corner, not its midpoint.

17. Continue interlace to all edges.

18. It helps to define the edges by drawing a lines around the perimeter of the corner.

19. The corner's outer edge will have a 180° turn in it; the inner sides will be two 90° turns.

### IV. Increasing the Width

20. Add another row of small squares to the grid of dots.

24. Leave a narrow rim of paper showing between the color inside the strand and the edge of the strand. You may further outline the color with a thin line of black ink, if desired.

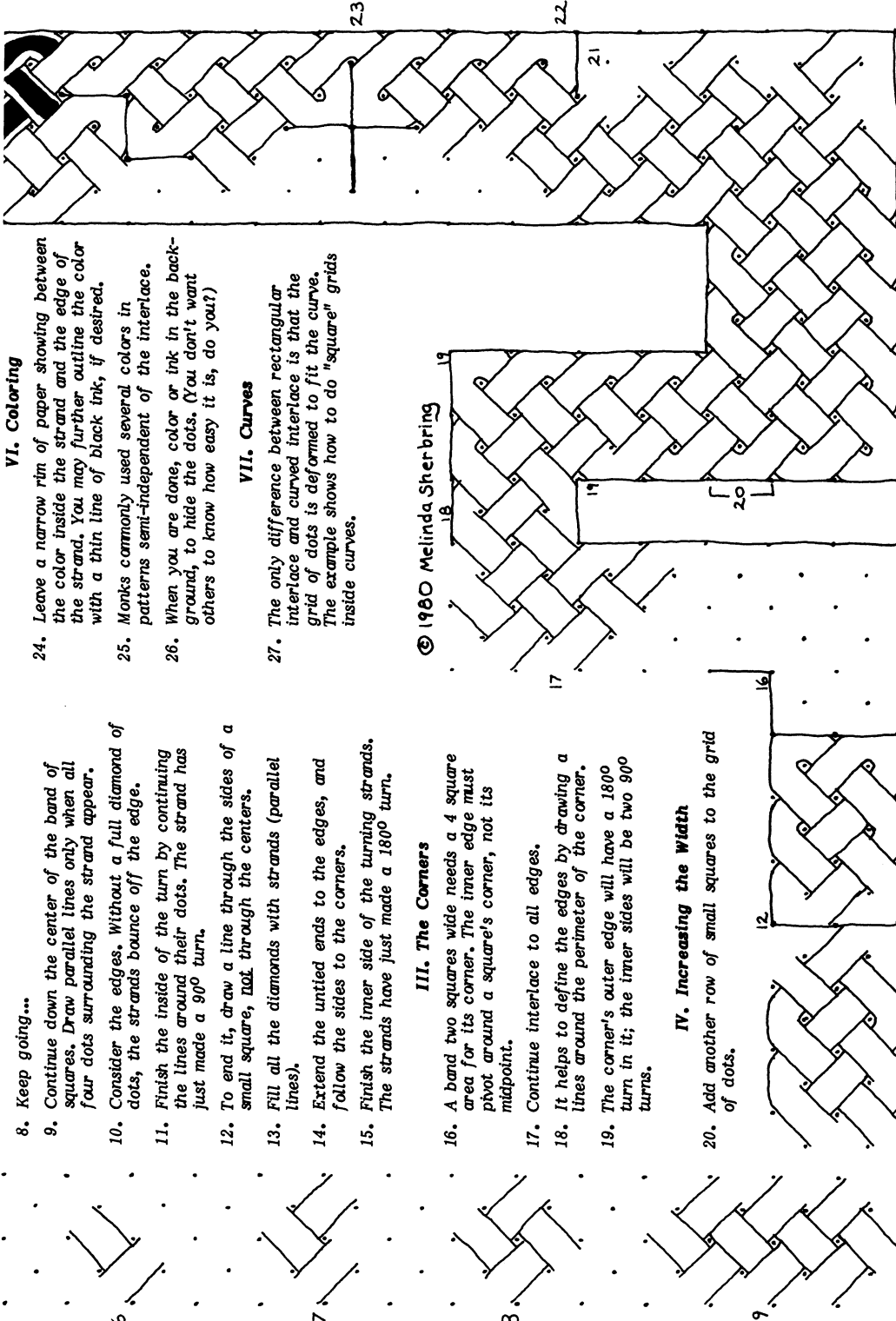
25. Monks commonly used several colors in patterns semi-independent of the interlace.

26. When you are done, color or ink in the background, to hide the dots. (You don't want others to know how easy it is, do you?)

### VII. Curves

27. The only difference between rectangular interlace and curved interlace is that the grid of dots is deformed to fit the curve. The example shows how to do "square" grids inside curves.

© 1980 Melinda Sherbring



Changing a break requires changing the indicators in the four items surrounding the break, and reprinting the four characters on the screen that surround the break.

### === Arrays ===

**BITS()** Tells which bit to change when setting breaks under various conditions.

**COLCHANGE()** Tells how much to change the column number of the arrow for various joystick positions.

**ROWCHANGE()** Like COLCHANGE(), but for the row number.

### === Variables ===

**BITAND** Contains the address of the bitwise-And USR() function.

**BITOR** Contains the address of the bitwise-Or USR() function. (BITOR actually isn't used in Knotwork. It was included originally because it might have been.)

**BITXOR** Contains the address of the bitwise-Exclusive-Or USR() function.

**BREAKS** Holds the address of the break table. Most changes to the table are made using POKes. To POKE the right place in the table, we have to know where the table starts.

**BREAKS\$** The break table itself. Except in setting up, the break table is referred to through POKes using BREAKS (above).

**BREAKSSIZE** The size of the break table.

**C** Holds some number related to the column number of the cursor. Exactly what it is changes from place to place.

**CH** Holds the new break-table entry while changing a break.

**CODE\$** Holds the actual machine code for the USR() functions.

**COL** Tells which column in the knotwork rectangle the arrow is in.

**CURSOR** Holds the arrow being used for the cursor. It may be one of two arrows.

**DELAY** Tells how long Knotwork should delay before responding to the joystick.

**DIR** Tells which direction (horizontal or vertical) the break should go.

**FILE\$** Holds the file name for loading or saving a pattern.

**H** Used in setting up the code for the USR() function.

**HASDOS** Notes whether there's a disk available on this particular Atari computer.

**HEIGHT** Controls the height of the rectangle containing the knotwork.

**I** Used for many purposes, but usually with FOR loops.

**IDLE** The counter used to delay response to joystick movements.

**J** Another variable of many uses. Also mainly FOR loops.

**L** Used in setting up machine code for the USR() function.

**M** Holds the row number of the upper-left corner of the square of four characters surrounding the break to be changed.

MISC\$ As its name implies, MISC\$ has no fixed use. It is used anywhere some string is needed, but which isn't important enough to warrant permanent space of its own.

N Holds the column number of the upper-left corner of the square of four characters surrounding the break to be changed.

OLDSTICK Tells where the joystick pointed last time we looked at it.

OLDTRIG Tells whether the joystick trigger (button) was pressed or not last time we looked at it.

P Points into the CODE\$ string while setting up the USR() function.

R Holds a number related to the row number of the arrow. How it's related varies from place to place.

ROW Tells which row the arrow is in, in the knotwork rectangle.

S Points into the break table while setting up the "bare canvas" of the knotwork rectangle, and while changing the break table.

T Used for many things, in many places.

UNDER Remembers what was on the screen where the cursor was put (what's under the cursor).

WIDTH Controls how wide the knotwork rectangle is.

#### === Constants ===

Cnnn Anything named C followed by a number holds that number. Those variables are used to save space in the program.

CHSET\$ Holds all eight sets of sixteen characters.

HORIZ One of the two directions breaks can be.

SCREENWIDTH The total width of the Atari's screen, in characters.

VERT The other of the two directions breaks can be.

#### ===== The Program =====

100-110

Set up various constants for later use.

120 Choose how big the knotwork rectangle is to be.

130 Go see if there are disks attached.

200 Figure out how much room we'll need for the break table.

210-220

Allocate all our arrays and strings.

230 Since the break table is going to be referred to mostly with POKes, find out where it starts.

300-330

Set up the USR() functions, and remember their addresses.



340-390

The machine code for the USR() functions. In assembler, they are:

FRO = \$D4			
BITAND:	BITOR:	BITXOR:	
PLA	PLA	PLA	;Throw away # of args
PLA	PLA	PLA	;Get high byte of 1st arg
STA FRO+1	STA FRO+1	STA FRO+1	;save it
PLA	PLA	PLA	;Get low byte of 1st arg
STA FRO	STA FRO	STA FRO	;save it, too
PLA	PLA	PLA	;Get high byte of 2nd arg
AND FRO+1	ORA FRO+1	EOR FRO+1	;diddle it
STA FRO+1	STA FRO+1	STA FRO+1	;and save result
PLA	PLA	PLA	;Get low byte of 2nd arg
AND FRO	ORA FRO	EOR FRO	;diddle it, too
STA FRO	STA FRO	STA FRO	;and save the result
RTS	RTS	RTS	;All done, so return

As you can see, the only difference between the three functions is the instruction used to do the actual bit twiddling. Everything else is identical.

410-440

Set up the tables of how the arrow should move for the various joystick positions.

800-835

Set up the eight sets of sixteen characters for later use in printing knotwork patterns.

850-880

Set up the table of which bits in the break table to change for various breaks being changed.

900 Clear the screen, turn off the cursor, remove the left margin, and put the knotwork font in effect.

910 Make the screen blank.

**{CLEAR} - Set up a blank knotwork canvas**

1000 Say that it'll take a short while to do.

1100-1180

Set up the basic pattern for the knotwork rectangle. The basic pattern is made out of four-character squares, selecting the first four character sets. Line 1120 sets up the characters that should be printed with sets 0 and 1. Line 1150 sets up the characters that should be printed with sets 2 and 3. No breaks are set anywhere. A dot is printed occasionally to reassure the user that all is well.

**Add the initial breaks around the edges**

1500 The characters along the left and right edges are selected from character sets 4 and 5. T holds the current character set number. Make S point to the first visible row of the rectangle (below the top edge).

1510-1570

For each row of the rectangle, put a break into the left and right edges of the rectangle. Also, mark that character as using one of sets 4 and 5. After doing the marking, switch the set used to the other of 4 or 5, for next time around the loop. Also, advance S to point to the next row in the rectangle.

1600 The characters along the top and bottom edges are printed using sets 6 and 7. Make T hold the first set (of 6 and 7) to use, and make S point to the first visible column of the rectangle. N holds the distance from the top to the bottom of the rectangle (just to save a little calculation in the loop).

1610-1670  
For each column of the rectangle, put a break at the top and bottom edges of the column. Mark each edge as using one of sets 6 and 7 when printing the characters on the screen. Then, switch T to the other set of 6 and 7, for the next time around the loop. Also advance S to the next column of the rectangle.

1700-1710  
Make the four corners of the rectangle all print using character set 4.  
**Print the initial rectangle on the screen.**

1900 For each row of the rectangle...

1910 Find the start of the row in the break table.

1920 For each column in the row...

1930 Put the cursor at the right row and column on the screen, and print a blank. (There may already be a pattern on the screen. We print a blank moving across the pattern to show that something is happening, even if the pattern isn't changing.) Move the cursor back to the right row and column again. (Alas, we can't just print a backspace. When you are printing to S:, backspaces are ordinary characters, and don't move the cursor backwards.)

1940 Print the proper character from the proper character set onto the screen. The break table has been arranged so that the number obtained by PEEKing at a spot in the break table is the number of the character to print from CHSET\$. No computations have to be done other than adding one to account for strings starting at character 1 instead of 0.

1950 Continue with the next column or row (whichever is appropriate).

2000 Start out at the upper-left corner of the rectangle.

2010 Assume that the joystick button isn't pressed, and that the joystick isn't pushed.

**The Main Loop - look at all the controls and do what's needed**

2100 Determine which character in the four-character square we are over.

2120 From that, figure out which way the arrow should point (and which way the break it points to goes).

2200 Remember what is obliterated by the arrow.

2210 Put the arrow up on the screen.

2220 Look at the joystick trigger. If it's moved since we saw it last, remember the new position. If the new position is **pressed**, go change a break.

2230 Look at the joystick itself. If it isn't upright, go move the arrow.

2240 Remember that it's upright now.

2250 See if there's a key pressed. If so, go do what it says.

2260 Nothing needed to be done. Go make the rounds again.

### **Move the arrow**

2300 If the joystick is in the same position as it was last time around, kill some time before responding.

2310 Time to respond. If the joystick is in the same position as it was before, cut down the delay for next time.

2320 If the joystick moved since last time, put the delay back at maximum.

2400 Put back what was erased by the arrow.

2410-2460

Move the arrow's position, but make sure it stays within bounds. Then go off and put the arrow back on the screen in its new position.

### **See what key was pressed**

3000 Put back what was erased by the arrow. (In this case, it tells that we've seen the keypress.)

3010 Strip off the SHIFT and CTRL bits from the keycode.

3020 If the key pressed is {RVS} or {CAPS}, change it to {ESC}, so we can ignore it noisily. (It's reassuring to hear the click when a key is pressed, even if the key is ignored.)

3030 Get the ATASCII code for the key. If it's in reverse video (black on white), change it to normal white-on-black.

3040 If the key is a lower case key, change it to upper case.

3110-3150

If the key is one of {CLEAR}, {Q}, {L}, or {S}, do the appropriate thing. Otherwise, ignore it.

### **L - Load a pattern**

3200 Go off and ask for a file name. If no name is given, don't bother trying to load; he's changed his mind.

3210 If anything goes wrong while loading, go to line 3300. Open the file he wanted read.

3220-3270

Read in the pattern, one character at a time. While we're reading it in, move a blank across the pattern to show how far we've gone. In the wake of the blank we put the new pattern.

3280 All is well. The pattern was loaded without problems. Close the file, and go back to the main loop.

3300-3310

Something went wrong. Tell him so. Give him time to read the message, and then erase it, to keep the screen tidy. Close the file, since it's of no use to us. Go back to the main loop.

### **S - Save a pattern**

3400 Go ask what file to put the pattern into. If he doesn't say, assume he didn't want to save anything after all.

3410 If anything goes wrong during the save, go to line 3500. Open the file to save the pattern into.

3420-3470

Copy the pattern into the file. While we're doing that, move a blank across the pattern to show how far we've gone.

3480 All done. Close the file, and go back to the main loop.

3500 It didn't work. Say so, and wait till he's had time to read the message. Erase the message, close the file (since we can't use it any more), and return to the main loop. (Everything after printing the message is actually done in line 3310. It was all listed here to keep the description straightforward.)

**Get a file name to load from or save to**

3600 Put whatever message is printed at the top of the screen. If we're dealing with cassette (no DOS, therefore no disk), go be nice to cassette people.

3605 "Press" the {CAPS} button, but remember what it was before. Print our question.

3610 Get his reply, and restore the caps-lock to whatever it was before. Erase the question (and reply).

3620 If he gave no name for a file, forget about doing anything.

3630-3650

If no colon (and therefore no device) was given, assume "D:".

3660-3680

If no period (and therefore no extender) was given, assume ".KNT" to keep knotwork patterns separate from other files.

3690 We now have a legal file name. Hand it back.

3700 Erase the top line of the screen for someone,

3800 Alas, he has only a cassette recorder. Ask if it's set up for whatever it's needed for.

3810 Get the reply (of one key), and erase the question.

3820 If he says "Y" (or anything except "N"), hand back a file name of "C:". (This routine was called to get a file name, after all.) Otherwise, hand back no name at all, to say that things aren't ready.

3830 We're done, so go back to wherever we came from.

**Change a break**

4000 Put back what was erased by the arrow.

4010 Turn off Attract Mode.

4020 Since a break is surrounded by a four-character square, figure out where the upper-left corner of the square is.

4030-4100

For each of the four characters in the square: If the character is outside the rectangle, ignore it completely. Find the character's position in the break table, and flip the appropriate bit for that break in that position. Go to the proper place on the screen, and print the character required to display the newly changed break.

4110 Now that we're done, go back to the main loop.

### **See if there's a DOS (and a disk) on this Atari**

4900 Assume that there's no disk.

4910-4930

Search the Atari's device table for a device named "D:". If there is one, assume it is a disk, and remember that there's a disk available. Once we've found out about the disk, return whence we came.

### **Q - Quit (leave the program)**

5000 Clear the screen, turn the normal cursor back on, and give control back to the user.

### **Read hexadecimal data into a string (for machine code)**

9100 Read one byte of data. If it's an equals-sign, we're done.

9110 Pick off the upper half of the byte, and change it to a number from 0 to 15.

9120 Do the same with the lower half.

9130 Combine the halves, and stow the result into wherever it belongs. Advance the store pointer, and go back for more.

---

# **Hacker's Delight**

## **A Compendium of Genuinely Useful Memory Locations**

A favorite pastime among hackers is to figure out exactly how the operating system (or OS) uses all the memory it keeps to itself. The usual result of such investigations is a memory map which tells in great detail what the OS does with each byte. What use any of those bytes are to the programmer is rarely explained. This map is different: it is in no way complete, but any location mentioned is explained in detail. It tells both what information is kept, and how you can put it to use. As such, the many locations which are of use to no one but the OS are ignored completely.

The first thing given for each location is the Official Atari Name for that location. We are using those names because they are a standard set of names, already in public use. (We see no reason to promote confusion by using names different from those Atari is using, even if we'd prefer different names here and there.) Immediately after the name will be the memory address, first in decimal, then in hexadecimal. If the name is for a multi-byte quantity, the range of addresses will be given. Except where otherwise stated, the low-order byte is at the low-numbered address. After the address will be a description of the location, with hints, examples, and caveats. (Some of these locations have to be used with a little care.)

This map has two parts: First is a list of locations in alphabetical order by name, with the addresses in decimal, and a very brief description of each location. Second is a list in order by address, with detailed expositions on the locations. The first list can be considered both a summary, and an index into the second list.

Since all the names below are names of memory locations, I will often talk of setting XXXX to something. What I mean is to POKE XXXX with the something, rather than assigning the something to XXXX. Similarly, if I speak of XXXX holding something, or being equal to something, I'm referring to PEEK(XXXX), not XXXX itself.

Some locations which are useful won't be described below because they require more explanation than can be given in a paragraph (usually because they control something big, with a need to set things up before you flip the switch).

The odd spelling of the various Atari names is due to the fact that the assembler they use for their software development can't take names longer than six characters.

In general, a single-byte location can be meddled with using PEEK and POKE directly. For example:

```
WIDTH = PEEK(RMARGN)-PEEK(LMARGN)+1
PRINT "Error number ";PEEK(ERRSAV)
POKE COLCRS,12
POKE SHFLOK,PEEK(KEYCODE)-60
```

Two-byte locations are more awkward, as you have to put the number together yourself (for PEEKs), or break it in half (for POKES). For example,

```
SCREEN=PEEK(SAVMSC)+256*PEEK(SAVMSC+1)
PRINT "Error at line ";PEEK(STOPLN)+PEEK(STOPLN+1)*256
CENTER=192
POKE COLCRS,ASC(CHR$(CENTER))
POKE COLCRS+1,INT(CENTER/256)
HI=INT(CENTER/256):LOW=CENTER-256*HI
POKE COLCRS,LOW:POKE COLCRS+1,HI
```

The POKE examples show two different ways to carving the number to be POKEd into two pieces. Both use INT(xxx/256) to get the upper half of the number. To get the lower half, the examples use two different methods. The first uses a trick: CHR\$() will take any (non-negative) number and use the remainder after dividing by 256. The ASC() then retrieves the remainder. The second example calculates the remainder directly.

(A note: in general, on the 6502, two byte numbers are stored with the low order byte in the lower numbered address, and the high order byte in the higher address.)

In the descriptions below, it is assumed that variables have been defined for any names that are used as memory addresses. If, when you use the names yourself, no values have been given to the names, nothing will work as you expect. All you need are a series of lines like this:

```
RTCLOK=18:ATRACT=77:ERRSAV=195:INVFLG=694
```

containing those names you're using in the program. Any locations you aren't using you can ignore completely.

#### The Locations by Name

ATRACT	77	Attract mode timer and flag. <128:off, >=128:on. On after 9 mins.
CHACT	755	Character mode. Bit 0=Blank RVS; Bit 1=Normal RVS; Bit 2=Flip char.
COLCRS	85	Cursor column number. Two bytes in GR.8.
CRSINH	752	Cursor inhibit. =0:visible, <>0:invisible

DSPFLG 766 Control-char display. =0:act, <>0:show  
 ERRSAV 195 Number of last TRAPped Basic error.  
 FILDAT 765 Color number for FILL (XIO 18)  
 FR0 212 Floating Register 0. Holds value for USR(). Is page zero RAM.  
 INVFLG 694 Reverse-video-input flag. =0:normal, =128:reverse, other:don't use  
 KEYCODE 764 Last pressed key. Not Atascii. =255:no key. Reset by GET, INPUT.  
 LMARGN 82 Left screen margin, 0-39. Used only w/ GR.0 and text window.  
 RAMTOP 106 Size of memory in 256-byte pages. Do GR.0 after changing.  
 RMARGN 83 Right screen margin, 0-39, >LMARGN. Used only w/ GR.0, text window.  
 ROWCRS 84 Cursor row number.  
 RTCLOK 18 Clock. PEEK(18) is slow, PEEK(20) is fast. Ticks every 60th sec.  
 SAVMSC 88 Address of upper-left corner of screen in memory.  
 SDLST 560 Address of start of screen display list.  
 SHFLOK 702 Caps-lock flag. =0:lowercase, =64:uppercase, =128:ctrl chars  
 SOUNDNR 65 Noisy I/O flag. =0:silent, <>0:noisy. Not a volume ctrl.  
 STOPLN 186 Number of line causing last TRAPped Basic error.  
 TABMAP 675 Bit map of tab stops, one bit per column.  
 TXTCOL 657 Text window cursor column number.  
 TXTROW 656 Text window cursor row number.

### The Locations by Address

RTCLOK (18-20, \$12-\$14) Holds the Atari's clock. It is updated every sixtieth of a second (a unit hereby christened a **tick**). Every sixtieth of a second, the Atari adds one to location 20. If the sum exceeds 255 (which is the largest number that can be held in one byte), location 20 is set back to zero, and location 19 gets one added to it. If location 19 exceeds 255, it is set to zero, and one gets added to location 18. If location 18 exceeds 255, that's the end of the line. Location 18 gets set to zero, and nobody gets one added. Notice that, unlike the usual arrangement of numbers on the 3502, the low bits of the clock are in the high byte of the number.

The main use of RTCLOK is, as you might expect, to time things. For example, to see how long a particular FOR loop takes to run, you might do the following:

```

100 RTCLOK=18:REM Tell Basic where the clock is
110 REM First, set the clock to zero.
120 FOR I=RTCLOK TO RTCLOK+2:POKE I,0:NEXT I
190 REM Next, do your FOR loop
200 FOR I=0 TO 39
210 POSITION I,0:PRINT "X"
220 NEXT I
290 REM Finally, get the elapsed time back.
300 N=0:FOR I=RTCLOK TO RTCLOK+2:N=N*256+PEEK(I):NEXT I
310 PRINT "Loop took ";N;" ticks, or ";N/60;" seconds."
320 END
  
```

RTCLOCK can be used to put delays into programs as well, but it is awkward enough to use that FOR loops (FOR I=1 TO 1000:NEXT I) are probably a better technique.

SOUNDR (65, \$41) SOUNDR lets you control whether or not sound comes out of your TV's speaker while the Atari is doing I/O to some device. If you type

#### POKE SOUNDR,0

the Atari will do its work silently. If you poke SOUNDR with anything else except zero, the Atari will make noise while doing I/O. In practice, I find that it is usually better to leave SOUNDR non-zero, and use the volume control. However, there are some applications in which the I/O noise is a nuisance, and ought be gotten rid of. SOUNDR allows you to do that. Note that SOUNDR is not a volume control; it is just an on/off switch.

ATTRACT (77, \$4D) In the short history of video games, it has been found that if you leave the same picture on a TV screen too long, the picture gets "burned" onto the screen. A burned-in picture has the unfortunate characteristic that it is always visible (even when the TV is off). The Atari people, being reasonably knowledgeable about video games, programmed their computer to avoid burned-in pictures. If you leave the Atari alone for about nine minutes, it will enter what Atari calls Attract Mode. In Attract Mode, everything on the screen changes color every so often. This protects the phosphors (which cover the front of the picture tube, and glow to make the picture) from being overused and wearing out (the cause of burn-in).

There are two ways to turn off Attract Mode. One is to press a key on the keyboard. (Pressing a button, like {START}, won't do.) The problem with this is that it requires the help of the person using the program. Also, only pressing a key will do. Fiddling with the joystick doesn't count. The other way to turn off Attract Mode is to POKE ATTRACT with zero. This can be done in the privacy of your own program, and requires no help from anyone. Also, it can be done at any time. Be aware that POKEing ATTRACT with zero doesn't shut off Attract Mode permanently. Nine minutes after the POKE, (assuming no keys are pressed) Attract Mode will go back on, unless you prevent it by POKEing a zero again before nine minutes go by.

A simple way to insure that Attract Mode stays off is to do the POKE whenever the user directs something to be done. For example, when he pushes the joystick, your program detects that, does whatever is to be done, and POKes ATTRACT with zero. That way, if he has actually walked away from the computer, Attract Mode will go on as usual, protecting the screen. As soon as he returns and does something, Attract Mode goes off again, leaving the screen in its original colors.

Should you, for some reason, want to turn Attract Mode on, you can do so by POKEing ATTRACT with 128.

LMARGN (82, \$52) and

RMARGN (83, \$53) LMARGN and RMARGN control the left and right margins of the screen. The margins are just the blank areas to either side of the text on the screen. The numbers stored in LMARGN and RMARGN are the column numbers of where the text starts and ends on the screen. (The columns are numbered starting at zero, so column zero is the leftmost column, and column 39 is the rightmost column.) The margins do not control scrolling; they just control where the text goes on the screen. When any scrolling



occurs, everything on the screen moves at once, be it inside or outside the margins. (Text can be printed outside the margins with the aid of the POSITION statement.)

The {DELETE LINE} and {INSERT LINE} keys are also affected by the margins. {DELETE LINE} and {INSERT LINE} always delete or insert 40-character chunks, which start at the left margin, and wrap around the edge of the screen back to the left margin again. The right margin is ignored completely.

When the margins are changed, the amount of text you can type in per logical line is changed. A logical line is always three (or fewer) physical lines, however long (or short) those physical lines may be. Moving the margins changes the length of the physical line.

Finally, changing the margins affects only the screen (not the printer), and then only in graphics mode 0, or in a text window.

ROWCRS (84, \$54) and

COLCRS (85-86, \$55-\$56) ROWCRS and COLCRS together specify the position of the cursor on the screen. As might be surmised, ROWCRS tells which row the cursor is in, and COLCRS tells the column. Of the two, COLCRS is the more useful. You can fake a TAB() function (which Atari Basic doesn't have) by POKEing the desired column number into COLCRS before doing your printing. (Remember that the Atari counts columns starting at zero.) If you want to change both the row and column of the cursor at the same time, you can POKE both ROWCRS and COLCRS, but using POSITION is both easier and clearer. However, if you want to find out where the cursor is, you have no alternative to PEEKing at ROWCRS and COLCRS, since the information isn't available any other way.

Be aware that, when you're in any graphics mode other than 0, ROWCRS and COLCRS tell the position of the graphics cursor, not the text cursor. (Because of this, POSITION, LOCATE, and PLOT all use the graphics cursor in modes 1 through 11, since they all change ROWCRS and COLCRS to do their work.) For modes 1 through 11, you have to POKE TXTROW and TXTCOL to move the text cursor. Also, in graphics mode 8, COLCRS is a two-byte number, since the column number can be greater than 255. To change (or find out) the column number, both bytes must be POKEd (or PEEKed). (The second byte is at COLCRS+1.) In any other mode, COLCRS is one byte long, and COLCRS+1 can be ignored.

SAVMSC (88-89, \$58-\$59) SAVMSC points to the start of screen memory itself, not to the display list. It is not often useful, as POSITION, PLOT, and LOCATE fill most possible needs. Occasionally, however, need arises to get something on the screen without letting the Atari fiddle with it in any way. You can bypass the Atari by POKEing to the screen directly:

```
SAVMSC = 88
SCREEN = PEEK(SAVMSC)+256*PEEK(SAVMSC+1)
FOR I=0 TO 959
  POKE SCREEN+I,0
NEXT I
```

Note that what you POKE to the screen does NOT use the ATASCII code.

RAMTOP (106, \$6A) RAMTOP holds the size of memory, measured in 256-byte "pages". If you change the number stored here, you will change how much memory

the Atari thinks it has. Theoretically, the Atari won't use any memory that it doesn't think it has. Why would I want to change the size of memory, you ask? Occasionally, especially when you play with ANTIC and CTIA (the Official Atari Names for the parts of the Video Wizardry chips), you need to grab chunks of memory that won't be in the way of anything, and which will stay around for a while. The easiest way to grab the memory you need is to tell the Atari that memory isn't as big as it used to be, and use the excess yourself. (Fontedit does that to hold the character generator.)

To reserve memory, PEEK at RAMTOP to find out how much memory the Atari has, subtract how much you need (plus some extra memory: see below), and POKE the result back into RAMTOP. Then, right after the POKE, do a GRAPHICS statement, even if you stay in the same mode. The screen memory is normally kept in the high end of memory, right where the memory you want is. In any conflicts in memory use, the Atari will win. The GRAPHICS statement removes the conflict by forcing the Atari to move screen memory to before the (new) end of memory. It is thus removed from the memory you want for yourself.

If every program you run needs a little private memory, you'll eventually run out of memory. Each program that runs will gnaw away at memory, since the memory grabbed is never returned. There are two solutions: either the program can return the memory when it's done with it (remember to give another GRAPHICS command!), or a program that grabs memory can grab it starting at the real end of memory, instead of starting at where the Atari thinks memory ends. It's simple to find out where memory really ends:

```
100 RAMTOP=106
1000 SIZE=PEEK(RAMTOP)
1010 ADDR=SIZE*256:T=255-PEEK(ADDR):POKE ADDR,T
1020 IF PEEK(ADDR)=T THEN SIZE=SIZE+1:POKE ADDR,255-T:GOTO 1010
1030 REM SIZE now holds actual size of memory.
```

Briefly, what happens up there is this: We find out how big the Atari thinks memory is. Next, we see if we can change the first byte beyond the edge. If we can, we change it back again, increase our idea of the memory size by one page, and go back and try again. Once we hit a place where we can't make the change stick, we've run out of usable memory, and we know how big memory is.

One major thing to keep in mind: that "theoretically" up there in the first paragraph. In practice, the Atari doesn't always confine its meddlings to below RAMTOP. When that part of the OS which clears the screen and does the scrolling was written, the people who wrote it made the assumption that the screen memory would always be the very last thing in memory. Working under that assumption, they took shortcuts in the code that oughtn't have been taken. As a result, when you clear the screen (using {CLEAR}), 64 bytes of memory immediately following the screen memory are cleared in addition to the screen. If you're going to use memory after the screen, you can either never clear the screen, or not use those 64 bytes immediately following the screen memory. Similar, but worse, is scrolling of a text window. When text is scrolled, 24 lines of text are moved, even though there are only 4 lines of text in the window. The other 20 lines (800 bytes!) are taken from the memory beyond the screen (your memory!). Needless to say, that sort of treatment does not do good things to what you're storing there. Again, your choices are to never scroll a text window

(tricky, but not very hard), or to not use those 800 bytes of memory. Faced with those choices, I've chosen to not use the 64 bytes, in the first case, and not scroll the text window, in the second. When I'm grabbing memory, I just grab an extra page, and waste most (or all) of it.

- STOPLN** (186-187, \$BA-\$BB) This location holds the line number of the line in which a TRAPPED error occurred. It is only changed when an error is TRAPPED. It can be printed out (along with the error number, see ERRSAV), or it can be tested ("did it foul up in line 43 again?"), or it can be used in a GOTO. The last is not recommended, as it leads to opaque programs, but it is legal.
- ERRSAV** (195, \$C3) This location holds the number of the error that caused the TRAP to be tripped. As with STOPLN, it is set by the Atari only when a TRAPPED error occurs. The number stored in ERRSAV is identical to the number that would have been printed in an error message without a TRAP.
- FR0** (212-217, \$D4-\$D9) FR0 is one of the "floating point registers" used by the Atari to do non-integer arithmetic in. It concerns the programmer mainly in how it relates to the USR() function. When a USR() function that is expected to return a result is called, it puts the result into the first two bytes of FR0 just before it returns. All six bytes are available for use by a machine-language program, as long as no floating point arithmetic is done by anyone. (Basic does nothing but floating point, so FR0 can't be used for storage of numbers between calls to the USR() function.) Note that FR0 is in Page Zero, with all that that implies. (For those not in the know, Page Zero is special when you're working in machine language. Using it can make your machine code smaller and faster.)
- SDLST** (560-561, \$230-\$231) The name SDLST was fashioned after Atari's names for the two bytes (SDLSTL and SDLSTH). They name each byte separately, while I see no need for more than one name. SDLST holds the address of the screen display list. The display list is a powerful thing to know about, and cannot be gone into here; I haven't the room. As this location is looked at every tick of the clock, you have to be very careful when changing it. If you don't take care, it will be looked at in a half-changed state. (Basically, you have to turn off one of the Atari's interrupts to be safe.) See Hacker's Delight in Iridis 1 for information on display lists.
- TXTR0W** (656, \$290), and
- TXTCOL** (657, \$291) TXTR0W and TXTCOL are analogous to ROWCRS and COLCRS, in that they tell the row and column of a cursor. However, TXTR0W and TXTCOL control the text cursor in a text window, whereas ROWCRS and COLCRS control the graphics cursor (or the text cursor in GRAPHICS 0). To move the cursor in a text window, you have no choice but to POKE TXTR0W and TXTCOL, because POSITION, PLOT, and LOCATE refer to the graphics cursor. Except for the change in names, everything described for ROWCRS and COLCRS applies to TXTR0W and TXTCOL.
- TABMAP** (675-689, \$2A3-\$2B1) TABMAP records where the tab stops are set on the screen. Each column on the screen has one bit in the table corresponding to it. (The table can therefore hold 15(bytes)x8(bits/byte)=120 tab stops.) If the bit for a column is on (1), a tab stop has been set at that column. If the bit is off (0), there's no tab stop. The most straightforward way to alter TABMAP is to use the {SET-TAB} and {CLEAR-TAB} characters. Space over to the column you want a tab set or cleared at. Then press {SET-TAB} or {CLEAR-TAB}, whichever is appropriate. (This can be done with PRINT statements just as easily.)

Some things are more easily done by diddling TABMAP directly, such as clearing all the tab stops. To do that, just POKE zero into all fifteen bytes of TABMAP. After the tabs are cleared, you can set new ones wherever you like, using {SET TAB}. Naturally, by POKEing in things other than zero, you can set up any combination of tab stops you like. For example,

```
FOR I=TABMAP TO TABMAP+14:POKE I,17:NEXT I
```

will set tabs at columns 4, 8, 12, 16, etc. across the screen (and clear any other tabs that may have been set).

```
FOR I=TABMAP TO TABMAP+14:POKE I,1:NEXT I
```

will restore the tabs to their original settings of columns 8, 16, 24, 32, etc.

When altering tab stops, keep in mind that tabs work with logical lines, not physical lines, and that a logical line can span up to three physical lines. Thus, if all logical lines on the screen were two physical lines long, with a tab set only at column 23, successive {TAB}s would place the cursor on every other physical line. The skipped physical lines would be those that held columns 41 through 80 of the logical lines. In practice, tab stops past column 40 are useful only if you know ahead of time how long the logical lines will be.

INVFLG (694, \$2B6) INVFLG is unusual in that it has no effect on the display. It affects only what is typed on the keyboard. INVFLG tells whether what's being typed is in reverse video (black-on-white) or not. If you POKE a number greater than 127 into INVFLG, anything typed will be in reverse video (black-on-white). If you POKE in 127 or less, things will stay in the normal, white-on-black style. POKEing INVFLG lets a program press the {RVS} key itself, without human intervention. If you change INVFLG yourself, remember that pressing {RVS} will change it, too. The exact behavior of INVFLG is this: after a keypress is converted to ATASCII, INVFLG is Exclusive-Or'd with the ATASCII code, and the result is handed off to the program that wanted input. Because of this, any numbers other than 0 and 128 in INVFLG will cause rather odd things to happen.

SHFLOK (702, \$2BE) SHFLOK is like INVFLG, in that it has no effect on the display. It, too, affects only what's being typed. SHFLOK corresponds to the {CAPS} key, and tells what kind of case lock is in effect. If SHFLOK is 64, {CAPS} was last pressed, and all letters are capitals-only. If SHFLOK is 0, {LOWR} was pressed, and all letters come in as lower case. If SHFLOK is 128, {CTRL}{CAPS} was pressed, and all letters come in as {CTRL}-whatever, resulting in the funny characters. You can POKE whatever you like into SHFLOK, thereby "pressing" whichever variant of {CAPS} you want. For example, to force typing in all caps (until {CAPS} is pressed in some form), POKE SHFLOK with 64. SHFLOK is used by the Atari while changing the keycode to the Atascii code. Partway through the conversion, if the key is an unshifted alphabetic key (i.e. 'a' through 'z'), SHFLOK is Logical-ORed into the keycode, and the resulting code is sent back through the conversion process. Normally, ORing in SHFLOK produces a control code or some such thing, which either isn't unshifted, or isn't between 'a' and 'z'. However, the keycodes are arranged such that if SHFLOK is between 1 and 63, ORing it with an unshifted letter keycode will produce another unshifted letter keycode. That keycode will in turn have SHFLOK applied, (producing the same code this time) and the Atari will go back through the conversion. Since the code will never change again, and it will always be an unshifted letter, the Atari will never finish converting the code to

Atascii. As a result, it will appear to "hang" (as we computer folk say), and won't respond to anything but RESET. Putting other numbers into SHFLOK will break the loop because the keycode will cease being unshifted after the first time around, and SHFLOK applies only to unshifted keys. The net result of all this is that you shouldn't POKE anything but 0, 64, or 128 into SHFLOK, as nothing useful results.

CRSINH (752, \$2F0) CRSINH controls whether or not the cursor is visible on the screen. If CRSINH is 0, the cursor is visible. If it is anything but zero (1, for example), the cursor is not visible. Keep in mind that the ONLY thing that CRSINH controls is whether the cursor is visible. The screen will still scroll if the cursor goes past the lower-right corner of the screen, even if the cursor is invisible. An invisible cursor moves in exactly the same way as a visible one. The main use for CRSINH is with images that get changed a lot; they look much better if there isn't an ugly white square wandering about the screen.

CHACT (755, \$2F3) CHACT controls the display of reverse video (i.e. any character whose ASC() is 128 or greater), and also one other thing. CHACT is divided up into separate bits, each on which controls one things. Bit 0 (which equals 1), if on, makes any reverse video character appear on the screen to be a blank. (This isn't necessarily invisible, as a blank can be either dark (when white-on-black), or bright (when black-on-white). Bit 1 (which equals 1), if on, makes reverse video characters come out as black-on-white. If it's off, they come out as white-on-black, so they look just like any other characters. The combined effect of bits 0 and 1 are summarized as follows: POKE CHACT,0 makes everything print as white-on-black. POKE CHACT,1 makes reverse video invisible. POKE CHACT,2 makes reverse video print as black-on-white. POKE CHACT,3 makes reverse video print as solid white blocks. When you turn on the Atari, it sets CHACT to 2. (Try the following program:

```
90 CHACT=755
100 PRINT "BLINKING TEXT"
110 POKE CHACT,INT(RND(1)*4)
120 FOR I=1 TO 100:NEXT I
130 GOTO 100
```

Type the underlined text in reverse video. Make sure you POKE CHACT back to 2 when you're done.)

Bit 2 (which equals 4) does something of no general use: when it's on, all printing comes out upside down. As I said, not very useful, but it is eyecatching.

One unobvious use for CHACT is to make text blink. By changing the number stored in CHACT at regular intervals, you can make reverse video text blink any way you like. The above program is a nice, albeit useless, example.

KEYCODE (764, \$2FC) KEYCODE is an invented name, to replace the Atari name of CH. CH is much too likely to be used as a real variable to make it a good name for a memory location. KEYCODE holds the keycode (not the ATASCII code) for the key last pressed. (That is, as long as the key isn't {BREAK}, {CTRL}, or {SHIFT}. {BREAK} won't show up in KEYCODE ever, and {CTRL} and {SHIFT} won't show up by themselves.) If no key has been pressed since the last time an INPUT or a GET was done, KEYCODE holds 255. Whenever a GET or an INPUT is done, the keycode is picked out of KEYCODE and

converted to ATASCII. KEYCODE itself gets 255 stuffed into it. (If KEYCODE is 255 when the GET or INPUT is done, the Atari waits around until a key is pressed to do something.)

The number in KEYCODE actually is more than the code for the pressed key. It also tells if {SHIFT} or {CTRL} was pressed with the key. If bit 7 (which is 128) is on, {CTRL} was pressed with the key. If bit 6 (which is 64) is on, {SHIFT} was pressed with the key. Naturally, both {SHIFT} and {CTRL} may have been pressed at once. In that case, both bits 6 and 7 are on at once. The other six bits (bit 0 through 5) hold the actual code for the key pressed. There's no particular order in the keycodes. That's no problem, as there are only two keys you're ever likely to need the codes for: {RVS} and {CAPS}. More on that later.

You can watch KEYCODE to make a program that keeps running until a key is pressed. Were you to do a GET to watch the keyboard, everything would grind to a halt until the user pressed a key. By watching KEYCODE instead, the program can keep doing things. Once KEYCODE stops being 255, the program can do a GET to find out what key was pressed. Or, if you don't care which key was pressed, just POKE 255 into KEYCODE.

One problem with that method is that the {RVS} and {CAPS} keys stop KEYCODE from being 255. However, they aren't enough to satisfy a GET. The GET will wait till some OTHER key is pressed. To avoid the wait, you've got to avoid doing the GET if the key pressed was either {RVS} or {CAPS}. The simplest test is to take the keycode from KEYCODE, and subtract 64 repeatedly until the code is 63 or less. (That removes the {SHIFT} and {CTRL} bits.) If the resulting code is 39, {RVS} was pressed. If it is 60, {CAPS} was pressed. In either case, you can either throw them away completely (POKE KEYCODE,255 and skip the GET), use them to change INVFLG and SHFLOK yourself (POKE INVFLG, 128-PEEK(INVFLG) for {RVS}, and POKE SHFLOK, PEEK(KEYCODE)-60 for {CAPS}), or change the KEYCODE to something else so that the GET will return without a wait (POKE KEYCODE,28 is handy; that makes GET return the {ESC} key). You can do some combination of the above, as well, such as handle SHFLOK, but throw away the keypress.

FILDAT (765, \$2FD) Controls the color of an area which is FILLED (XIO 18). FILL is similar to DRAWTO in that both FILL and DRAWTO cause a line to be drawn from one place to another. The difference between a FILL and a DRAWTO is that the FILL fills the area to the right of the drawn line with (possibly another) color. Also, doing a FILL is more complicated than doing a DRAWTO.

To do a DRAWTO, you'd need to execute

COLOR code : PLOT oldx, oldy : DRAWTO newx, newy

Code is the color number (not the color register number) for the color of line you want. Oldx and oldy are the starting position for the line. (The PLOT isn't needed if you want to draw a line from the last position you were at.) Finally, the DRAWTO draws the line to the new position.

To do a FILL, you would have to do the following:

COLOR code1 : PLOT oldx, oldy : POSITION newx, newy  
POKE FILDAT, code2 : XIO 18,#6,0,"S:" : PLOT newx, newy

Code1 is the color number for the line to be drawn. The PLOT sets the

starting position for the drawn line, as it does with DRAWTO. (Again, if you want to draw from the last place you were, you can omit the PLOT.) Next, you POSITION the cursor to the other end of the line. POSITION doesn't draw the line, it just tells where the line will go. Next, POKE FILDAT with the color number for the color to FILL the area to the right of the line with. Then do the XIO, as written. The line will be drawn and the area filled with color. Finally, the last PLOT is needed to position the cursor properly, as there is a bug in FILL which leaves the cursor in the wrong place.

One more note on FILL: when you do a FILL, the IOCB (or I/O channel) is left write-only. In other words, you can't do a LOCATE to read the screen after you do a FILL using channel #6. The solution is to use some other channel to do the FILL with. For example, you can write

```
XIO 18, #7, 0, 0, "S:"
```

to do the FILL using channel #7 instead of #6. (Naturally, you shouldn't use a channel you're already using for something else.)

DSPFLG (766, \$2FE) DSPFLG determines how screen function characters (such as {CLEAR} or {UP}) are handled. Usually, they are obeyed (by clearing the screen, or moving the cursor, or something). You can force them all to be displayed as the corresponding symbols by POKEing DSPFLG with 1 (or anything other than zero). To allow the function characters to do things again, POKE a zero into DSPFLG.

Page 6 (1536-1791, \$600-\$6FF) This is a page (256 bytes) of memory which Atari solemnly swears that they will never use for any purpose whatsoever. As such, it is guaranteed to be forever available to any program that wants to keep things in it. It can be used to hold tables, machine code, player-missile bit maps, betting odds, stock prices, pork belly futures, or whatever you like.

---

## Loadfont

If you are going to write programs using private fonts, you need a way to insure that your font is loaded. The Loadfont subroutine provides just such a way. It checks to see what font, if any, is loaded. If it isn't the one you want, it proceeds to load yours. (Knotwork uses a variant of Loadfont to make sure that the Knotwork font is loaded.)

To use Loadfont, put it in as lines 30500-30700 of your program. Change the assignment to F\$ (in line 30500) to the name of your font (leave the D: in the string there). Finally, add a GOSUB 30500 as the first thing done in your program. That's all that has to be done.

Loadfont is written to be entirely self-contained, even to dimensioning the string. If you want to change variable names (especially the string) to names more in keeping with your style of naming things, feel free. Also, it can be moved to a different range of lines without problems. Just make sure you get the GOTO's and THEN's renumbered properly. Also, Loadfont must always be run before you put anything on the screen of any value, because it changes the graphics mode of the screen while loading the font.

There are two simple ways to make Loadfont part of a program. The first way is to load Loadfont before you start typing in your program, and just write your program around it. The other is to keep Loadfont in text form (using LIST to a file), and

ENTER it when you discover you need it. ENTER's behavior is easy to explain: it makes Basic think that you are typing in statements, even though the stuff is really coming from a file.

An example will help make things clear: You have written a program to serve as a canvas for doing knotwork (sound familiar?) and you need to make sure that the font you created for it (using Fontedit, of course) is loaded into memory before anything else is done. You think, "That's why they wrote Loadfont!". Having LISTed it to your disk (or cassette, as the case may be) beforehand, you type

ENTER"D:LOADFONT"

Next, "FANCY" in line 30500 gets changed to KNOTWORK. (.FNT stays, because that marks it as a font.) Next, line 90 (GOSUB 30500) is added, your program having started at line 100. Finally, you save the modified copy of your program. (If you don't, Murphy's Law decrees that you'll forget to completely, and have to do all that again.) The final result of all that work is the following:

```

90  GOSUB 30500
100  REM The Great, Glorious Knotwork program starts here
110  REM Written by PHILANDER C. KNOX
    .
    .      (The Knotwork program itself)
    .
9999  END
30500 P=PEEK(106):DIM F$(20):F$="D:KNOTWORK.FNT"
    .
    .      (The Loadfont routine)
    .
30700 RETURN

```

The example assumes that you have a disk. If you don't, just change "D:" to "C:"; everything else stays the same.

## Loadfont Listing

```

30500 P=PEEK(106):DIM F$(20):F$="D:FANCY.FNT"
30510 A=P*256:C=255-PEEK(A):POKE A,C:IF PEEK(A)=C THEN POKE A,255-C:P=P+1:GOTO 30510
30520 P=P-5:IF P<PEEK(106) THEN POKE 106,P:GRAPHICS 0
30530 P=P+1:A=P*256
30540 IF PEEK(756)<>P THEN 30600
30550 I=LEN(F$):J=A-I-1
30560 FOR T=I TO 1 STEP -1
30570 C=ASC(F$(T)):IF PEEK(J+T)<>C THEN 30600
30580 IF C=ASC(":") THEN 30700
30590 NEXT T:GOTO 30700
30600 TRAP 30610:OPEN #1,4,0,F$:GOTO 30630
30610 IF PEEK(195)=130 AND F$(1,1)="D" THEN PRINT "PUT TAPE WITH ";F$;" INTO CASSETTE.":
    F$(1,1)="C":GOTO 30600
30620 PRINT "CAN'T FIND ";F$;".":END
30630 TRAP 30650:PRINT "{CLEAR}LOADING ";F$:C=ASC("X"):POKE 752,1
30640 FOR I=0 TO 1023:GET #1,J:POKE A+I,J:PRINT CHR$(C);"{LEFT}";:C=107-C:NEXT I:
    CLOSE #1:GOTO 30660
30650 GRAPHICS 0:PRINT "CAN'T LOAD COMPLETE FONT.":POKE A-1,0:END
30660 I=LEN(F$):J=A-I-1
30670 FOR T=I TO 1 STEP -1
30680 C=ASC(F$(T)):POKE J+T,C:IF C=ASC(":") THEN 30700
30690 NEXT T
30700 POKE 756,P:RETURN

```



Loadfont is concerned first, with making room for the character generator, and second, with loading the font into the generator.

### **=== Variables ===**

All the variable names are short to cut down on the size of the code, and to make it more likely that they're reused by the program calling Loadfont.

Also, many of the variables in Loadfont are reused whenever possible, again to keep it small and inconspicuous. The ones described below are those variables with specific uses.

- A Contains the address of the real end-of-memory.
- F\$ The name of the file that the font is stored in.
- P Holds the size of memory, in pages.

### **=== The Program ===**

#### **Find (or make) room for the font to load**

- 30500 Reserve room for the name of the file that the font is kept in, and put the name in it. Find out where the Atari thinks memory ends.
- 30510 See if memory really ends there. We do so by seeing what's there, and putting something else in its place. If, when we look again, what we put there is still there, we haven't found the end of memory. Put the old value back, and check one page farther on.
- 30520 We've found the real end of memory. Make sure that we get the last five pages of it, by blocking the Atari out of it if need be. (We do that by moving where the Atari thinks the end of memory is to before the memory we want ourselves.) The GRAPHICS 0 is to make sure that the screen memory isn't in the memory we want.
- 30530 Establish where the new character generator starts.

#### **Load the font into the character generator**

- 30540 There may already be a font loaded into our generator. If not, go directly to where we load the one we want.
- 30550-30590
  - There is a font already loaded. See if it's the one we need. We can tell because the font name is stored in memory just before the start of the font itself. We just compare the name of the loaded font with the name of the font we want. If they're the same, we have nothing to do. If not, we must load the font we need.
- 30600 Open the file that the font is stored in. If it's there, go read it in.
- 30610 It wasn't there. If that's because we wanted it on disk, and there was no disk, try to get it from cassette.
- 30620 No dice. Complain, then stop.
- 30630-30640
  - Copy in the font. If all is well, we'll have no problems. While we're copying, we print something that shows that we're hard at work. When we're done, store the font name to show what's been loaded.
- 30650 Alas, we can't read in the whole font. Since we need it to do anything, complain and go away. Before going away, make sure that no one else will think a font is loaded, by wiping out part of the name of the loaded font.

30660-30690

Store the name of the newly loaded font just before the font itself. Everything from the colon on is saved. (The colon is saved to remove confusion between similar names. Without the colon, FIN.FNT might be thought to be loaded when ELFIN.FNT was. With it, :FIN.FNT can't be confused with :ELFIN.FNT.)

30700 Make the Atari use our font, and return to whoever called us.

---

## Oddments

ODDMENTS is the repository for all the facts, fancies, and rumors that don't warrant an article of their own. We encourage contributions, and will acknowledge anything we see fit to use. Here's your chance to see your name in print.

--

One of the Atari's nifty features is the four-voice sound. When you use it, be careful what sort of I/O you do. If you talk to anything that hangs off the serial bus, the sound registers will be turned off. This happens because part of the circuitry that makes the sound is also used to run the serial bus. After you do any I/O, you'll have to reset the sound registers to what you want. In practice, I/O and sound aren't used together often, so it's not a big problem. Also, I/O to the screen and the keyboard have no effect on the sound registers. It's something you should be aware of, just in case. (One less mysterious bug to track down with days of sweat.)

--

After you execute a TRAP statement, the next error to occur sends you to the stated line number. The pitfall with that is that the TRAP remains in effect even after the program stops running. Should you give a command that is in error (trying to load a nonexistent file because of a mistype, for example), Basic will obediently restart your program at the appropriate line. This can cause rather odd effects at times. Also, unless the program prints out the error number, you can't find out what the error was. The only solution we know of is to turn off any TRAP after you're done with it (with TRAP 50000), and especially right before an END statement.

--

One of our readers (**Steve Steinberg** of Reston, Va.) sent in a different chime for the CLOCK program in Iridis 1. Unfortunately, it won't work well with CLOCK itself, since CLOCK can't afford to spend time in delay loops; it has to be constantly updating the screen to keep the second hand moving. However, the chime does sound nice. Here it is:

```
100 FOR Z=12 TO 1 STEP -1
110 SOUND 0,10,2,Z
120 FOR W=1 TO 40:NEXT W
130 NEXT Z
140 FOR W=1 TO 600:NEXT W
150 GOTO 100
```

--

In Iridis 1, we described something that the Atari does as "Glitch Mode". We have since found out that the official name for that is "Attract Mode". We have also found out why it's called Attract Mode, since it clearly won't attract people to the Atari. Way back when, before Atari began making computers, they were making video

arcade games. Arcade games, to make money, have to attract people to play them. Therefore, they were made with two "modes": play and attract. Since the arcade games spent 90% of their time in attract mode (or so we are told), the attract mode pictures would get "burned in" on the screen if nothing were done to prevent it. "Burn-in" occurs when the phosphors on the screen get over-used, and wear out. (The phosphors are the part of the picture tube that glow, and make an image by glowing.) So, attract mode did various and sundry things to preserve the phosphors. With the advent of the color video arcade game, the phosphor problem became more complicated: there were three different phosphors (for red, green, and blue) on the screen, and all three had to be kept intact. Atari chose to solve the problem by switching colors every so often. The color-switching function later became known within Atari as the "attract function", since it was associated with the arcade game attract mode. The original concept, of attracting people, ceased being the central idea. When the Atari computer was made, it, too, incorporated a color-shifting function to keep the phosphors working. By analogy with the arcade games, the mode in which the Atari did color shifting was called "Attract Mode", since it performed the "attract function".

--

An update on the Saga of the Break Key: As you recall from the last Iridis, we mentioned in passing that the Break Key isn't all that it could be. Ideally, Break would cause an error just like all the other errors. In particular, if you had a TRAP statement in effect, pressing Break would cause Basic to go off to the line named in the TRAP. This would allow better user-proofing of programs, as well as providing for a "Cancel that last command! I've changed my mind!" type of thing. In the absense of the ideal, the next best thing would be to be able to make the Atari ignore the Break key entirely. That way, pressing Break wouldn't cause any harm, even if it didn't do any good. As it stands, pressing Break kills the program, leaving its affairs in a rather disordered state. (Basic itself is still in fine shape; it's just your program that isn't well.)

Since the last Iridis, we've come across a possible method for making the Atari ignore the Break key. (Still no way to TRAP it, though.) Location 16 in memory controls, among other things, the BREAK KEY INTERRUPT. If bit 7 of that location is set to zero, the Atari won't recognize that Break has been pressed. (Meddling with the other bits does odd things, so we don't recommend playing with it unless you know the rules of the game.) So far, we've been able to disable Break, but the problem is that it doesn't stay disabled. Somewhere in the guts of the Atari is a piece of code that says, in effect, "Gee, someone turned off Break. That won't do at all. I'll turn it back on." We don't know how, when, or why that piece of code is called upon to do its duty. So, as things stand, there's a way to turn off Break, but only temporarily, and for an unknown amount of time. It is not yet information that is generally useful.

--

As you know, the Atari computer has nine graphics modes available to it, numbered 0 through 8. What you may not know is that it also has graphics modes 9, 10, and 11. That is, the OS knows about those modes. The hardware doesn't, and therein lies a tale. It seems that when the Atari computer was due to be released, Atari hadn't gotten all the bugs out of their super video chip (the GTIA), so they put out the machine with an "impoverished" version, called the CTIA. (You can't do things much fancier than Star Raiders. Would that other computers could be so impoverished!) However, the people who wrote the OS assumed that the GTIA would be available, since Atari had told them that it would be. Thus, the software can support things that the hardware can't do.

We have reason to believe that, in the time since the Atari computer was unleashed on the world, Atari has made the GTIA work. The GTIA is what you need to make graphics modes 9, 10, and 11 work. All three modes give you 80 dots horizontally (each a half-character wide) by 192 dots vertically (160 with a text window). Mode 9 allows one color on the screen, in sixteen different brightnesses. Mode 10 allows nine different color/brightness combinations, controlled by nine color registers. Mode 11 lets you have sixteen different colors, all of the same brightness. Again, modes 9 through 11 do NOT work on the Atari computer as is. You need a GTIA.

At present, Atari hasn't decided what to do with the GTIA. There are rumors (repeat, rumors) that they're going to drop it in favor of a still more powerful chip, yet to be described. Other rumors say that the European version of the Atari is being sold with a GTIA. (The video chips had to be redesigned to work with the PAL system, and it's as easy to redesign the fancy version as the plain one. Or so the rumors go.) We suggest that you just remain patient for now. More information about the GTIA and how to get one will be available eventually, and we'll print it as soon as we get it. Until then, find out what the CTIA has to offer. We can assure you that the stand-in chip you have in your Atari can do wondrous things, the likes of which you won't see anywhere else. It will be a while before you need anything more powerful.

--

A very nasty bug in string handling in Atari's Basic has surfaced. When the length of the string being copied is a multiple of 256 bytes long (i.e. 256, 512, 768, 1024, etc.), the correct number of bytes are copied, but they are copied to a place 256 bytes beyond the place where they should go. For example, assume that A\$ is dimensioned to 256 characters, and B\$ to 512. Then, B\$=A\$(1,256) will copy A\$(1,256) into B\$(257,512), not B\$(1,256). However, B\$'s length will be 256. The apparent effect is that the assignment didn't get done. Worse still is this: A\$=B\$(1,256). As before, B\$(1,256) will get copied into A\$(257,512). However, in this case, A\$ is only 256 bytes long. Therefore, the copied string will overwrite whatever it is that follows A\$ in memory. That will produce some very mystifying effects. Sometimes. Other times, nothing visible will happen.

What makes this bug truly insidious is that you rarely know ahead of time how long the strings you're copying will be. It's rather difficult to work around the bug under those circumstances. You can, of course, always use strings of 255 bytes or less. (Most people do, in fact. That's why the bug didn't show up for a while.) However, that throws away one of the nice features of Atari Basic strings: they can be extremely long. Alas! They just don't write interpreters like they used to.

---

## The Oracle

We have felt for some time that we ought to have a place where you can send your questions about the Atari computer. Since we didn't trust just anyone to answer your questions, we undertook a long and arduous quest in search of exactly the person we needed. We are happy to report that the quest was successful. Our pleas have been heard by the Oracle of Atari. He (or perhaps she - the Oracle is very secretive) has consented to answer inquiries from our readers. (Naturally, being an Oracle, she (he?) disdains mere employment by any company, be it us or Atari.)

The first question comes from **Steve Steinberg** from Reston, Va. His question: "I know that you translate Microsoft Basic's INPUT A(X) into Atari Basic INPUT T:A(X)=T. But how do you translate INPUT A(X),A(Y)? Also, how do you translate INPUT A\$(X),A\$(Y)?"

**The Oracle's** response: The translation for your second INPUT can be divined by methods closely allied to those you used for the first INPUT. The correct translation for INPUT A(X),A(Y) is

INPUT T1,T2 : A(X)=T1 : A(Y)=T2

You must use as many variables as you have array elements, and then do all the assignments afterwards.

I grieve to report that there is no direct translation of the string INPUT, for Atari Basic has no such thing as arrays of strings. You will have to rewrite the program so that it doesn't need them.

--

**Richard Auclair** of San Diego asks: "How do you use the GET statement? I'm told there is one, but every time I try to use it, I get errors."

**The Oracle's** response: Your problem is that you are neglecting to OPEN a file for the keyboard before attempting the GET. (A common error, as the need for the OPEN is obvious only to those who have been using computers for milleniums, as I have.) What you should do is:

OPEN #7, 0, 0, "K:"

The two zeroes are mystic symbols; they must be there, but in themselves they have no effect. The 7 is made an alias for the keyboard, another name by which you may invoke it. The "K:" is the name of the keyboard.

Later, when you wish to read a single keypush, you must write

GET #7, T

The 7 is the alias set up by the OPEN. The ATASCII code for the key pressed is put into the variable T. (Naturally, as the name for the thing is less important than the thing itself, any name may be used in a GET. You need not confine yourself to using T.)

Be aware that some keys cannot be bound by the GET statement: GET will never return a code for {RVS}, {CAPS}, {CTRL}1, {CTRL}3, or {BREAK}. The keys {RVS}, {CAPS}, and {CTRL}1 will slip through the clutches of the GET. {CTRL}3 and {BREAK} will defeat GET entirely: {CTRL}3 causes an error (number 136, which may be TRAPped); {BREAK} causes the program to cease execution completely.

---

## Novice Notes

Bit patterns. It's enough to make a grown-up frown, or at least yawn. What could be more boring and less useful than bit patterns? After all, the Atari is a well-designed Home Computer, right? And we all know that Joe and Sally Six-Pak shouldn't need to deal with nasty technical details like patterns of bits in a computer. And neither do you, most of the time. The Atari has a fine operating system, and a nice version of Basic. The folks that designed the system tried pretty hard to hide the jagged edges of computers from you. They almost succeeded, too! There are high-level statements or operating system calls for most everything you'll ever want to do with the machine. But there is a limit to how much a moderately priced computer system can provide, since there is only so much room in the ROMs. I'm sure there were features that they would really have liked to put in, but there wasn't space. My guess is that in the future we'll see a completely new 16K Basic from MicroSoft for the Atari. Maybe we'll get lucky and be blessed with a good high-level language such as Pascal too!

As good as the Atari system is, there are still many things the hardware can do that the current software doesn't support at a high level. As a result, you'll need to know a little about bit patterns if you want to do some of the advanced stuff that can be done with an Atari. (A bit pattern is just a collection of ones and zeros.) Please don't misunderstand: we'd very much like to be able to ignore these details ourselves. Someday, computers will be much more powerful and also much easier to use than they are now. In the meantime, if you want to get everything out of a computer you have to be willing to learn more about the technical details than you might wish.

A simple example of something that can't be done without a little knowledge of bit patterns is the way that you change how reverse-video text is displayed. (Reverse-video is when the text is black-on-white instead of the normal white-on-black.) As we discussed on page 44, you can change the way that reverse video is displayed, but to do so you have to POKE location 755 with a value from 0 to 3. There are quite a few other hardware features that you can control by POKEing certain locations with the appropriate bit pattern. This is especially true of the video chips. But before we can go much further with this topic, we need to define a few terms.

You probably already have a pretty good idea of what the PEEK and POKE statements in Atari Basic do: PEEK tells you the value stored at a given place in memory. POKE is the opposite of PEEK: it puts a new value at a specific memory location. For example, you might type:

```
PRINT PEEK(755)
```

which will print the contents of location 755 as a number. If you want to dink around, here's a tiny little program that makes it easy to see what is stored at various places in memory:

```
100 PRINT "WHICH LOCATION? ";INPUT LOC
110 PRINT PEEK(LOC);" {ESC}"; CHR$(PEEK(LOC))
120 GOTO 100
```

(The reason we print an ESC character right before we print the character value with CHR\$ is that a few of the Atari's characters have special meanings (for example, the keys that you use for editing the screen) as well as representing printable graphic characters. As you recall, the way to make these special characters "behave" is to print an ESC character in front of them. The trick is that it is OK to print an ESC in front of all the other characters too, since it doesn't affect them!)

And just what is a "location in memory"? Well, the memory of all computers is divided into a bunch of pieces. Each piece is called a "memory location", and has a unique number associated with it that is called its "address". You might imagine a very long street with houses all along it. When you first enter the street the very first house has a street address of "0", the next has the address "1", and so forth. On your Atari, at the far end of the street the very last house would have the address 65,535. This means that your Atari (which uses a microprocessor chip called the "6502") is able to talk to over 65,000 different memory locations.

Now we know how many memory locations are possible in an Atari. (Remember though, that the designers of the Atari decided that you can only plug in 48K of

RAM memory.) What is stored at each of these memory locations is something called a "byte". (Actually, the location is also called a byte.) And what is a "byte"? Well, byte is a term that seems to have originated somewhere in IBM when the IBM System 360 was designed back in the early 1960s. It is used to refer to a collection of eight bits. A given location in the memory of your Atari holds one byte, which is a collection of eight bits.

And what is a bit? It is a binary digit, which is either "1" or "0". (This seems to go on and on: every definition requires another term that also needs definition!) "Binary" refers to a number system that uses a base of two. We could devote an entire Novice Notes article to explaining various number systems. This isn't it.) Binary is convenient for computer hardware, as digital electronic logic typically can represent two states "ON" and "OFF".)

When you read our Hacker's Delight article in this issue you'll notice that we often need to talk about specific bits within a byte. By convention, the rightmost bit (also called the "least significant" bit) is number zero, and the leftmost or "most significant" bit is number seven. Here is a picture that also shows the decimal value of each bit in a byte:

Bit number:	7	6	5	4	3	2	1	0
Decimal value:	128	64	32	16	8	4	2	1

Here are a few examples that show bit patterns and their decimal equivalents. Please realize that we are only trying to illustrate The Big Idea. If we made this table complete there would be 256 different patterns below.

Bit Pattern:      Decimal value:

0000 0001	1
0000 0010	2
0000 0011	3 (2+1)
0000 0100	4
0000 0101	5 (4+1)
0000 0110	6 (4+2)
0000 0111	7 (4+2+1)
0000 1000	8
0000 1001	9 (8+1)
0000 1010	10 (8+2)

<u>Bit Pattern:</u>	<u>Decimal value:</u>
0001 0000	16
0001 0010	18 (16+2)
0110 0000	96 (64+32)
1000 0011	130 (128+2+1)
0000 1111	15 (8+4+2+1)
1111 0000	240 (128+64+32+16)
1111 1111	255 (128+64+32+16+8+4+2+1)

As you can see, it isn't very difficult to figure out the decimal value that represents a desired bit pattern. First, you decide which bits you want to change, and then look at our table above to find the decimal values for those bits. For example, if we want to make bit 6 a "1", we look at the table and find that bit 6 has the decimal value 64. But that's too easy! Let's say we want to put a "1" in bits 6 and 3. We look in the table and see that bit 6 has the value 64 and bit 3 has the value 8. We add 64 and 8 and find that our value for the POKE statement is 72.

Sad to say, there is trouble in paradise. Quite often you will want to flip certain bits in a given memory location while leaving the other bits undisturbed. Unfortunately, Atari 8K Basic doesn't have a way to do this. For bit-twiddling, you really need functions in Basic that do "bitwise" operations, especially logical OR and logical AND. If you have two 8-bit values, when you "OR" them together the result will have a "1" in any bit position where either of the two original values had a "1". For example, when we OR together the bit pattern 00000001 with the pattern 00000100 the resulting pattern will be 00000101.

The AND operation puts a "1" in those bits that are "1" in both of the patterns fed into it. For example, if we AND together the values 11110000 and 11000000 the result will be 11000000, since bits 6 and 7 are "1" in both of the values we ANDed together.

What to do? Well, the only answer is unpleasant indeed. We have to resort to machine language, and write those missing routines ourselves. Guess what? You've come to the right place! We provide the missing routines in the Knotwork program in this IRIDIS. The routines are listed in the DATA statements in lines 340 through 390. The routine that READs those DATA statements and stuffs the values away is at lines 9100 through 9130 of Knotwork.

Once the machine code is stored, we can use those functions with the USR function in Basic. An example of using the bitwise AND is shown at line 2100 of Knotwork where the variable **R** is assigned the bit pattern that results from ANDing together the row and the value 1.





























# About IRIDIS Listings

Our notation for program listings is based on two rules: anything underlined is reverse video, and anything in braces is special. Anything else is just what is appears to be.

Special characters in braces come in two flavors: single characters or words. Single letters or punctuation marks represent control characters (the ones you get when you press a key while holding the CTRL key down). For example, in IRIDIS listings, {C} is CTRL-C, and {,} is CTRL-comma (which prints the "heart" symbol). A word within braces is the name for a key. For example, {CLEAR} means the key labelled "CLEAR". A number before a letter or a word tells how many times to press the indicated key. For example, **PRINT "{CLEAR 10 DOWN}HI THERE"** will clear the screen, go down 10 lines, and print "HI THERE". To type that in, you would refer to the table below, which says that {CLEAR} is entered by pressing ESC followed by the CLEAR key. "10 DOWN" means the sequence of ESC followed by CURSOR-DOWN (or CTRL-equals), repeated ten times.



















When we actually want to print a down-arrow, and not do a cursor-down, our listings will show **PRINT "{ESC DOWN}"**. Using the table, we see that {ESC} means "press the ESC key twice", and {DOWN} means "press ESC, and then press cursor-down (ctrl-equals)". Put these together, and you will press ESC three times, and then press cursor-down.

Atari: Us: You type:

	{A}	ctrl-A
	{B}	ctrl-B
	{C}	ctrl-C
	{D}	ctrl-D
	{E}	ctrl-E
	{F}	ctrl-F
	{G}	ctrl-G
	{H}	ctrl-H
	{I}	ctrl-I
	{J}	ctrl-J
	{K}	ctrl-K
	{L}	ctrl-L
	{M}	ctrl-M
	{N}	ctrl-N
	{O}	ctrl-O
	{P}	ctrl-P
	{Q}	ctrl-Q
	{R}	ctrl-R
	{S}	ctrl-S
	{T}	ctrl-T
	{U}	ctrl-U
	{V}	ctrl-V
	{W}	ctrl-W
	{X}	ctrl-X
	{Y}	ctrl-Y
	{Z}	ctrl-Z

Atari: Us:

You type:

	{,}	ctrl-comma
	{.}	ctrl-period
	{;}	ctrl-semicolon
	{BACK}	ESC BACK
	{BELL}	ESC ctrl-2
	{CLEAR}	ESC shift-less
	{CLR TAB}	ESC ctrl-TAB
	{DEL CHAR}	ESC ctrl-BACK
	{DEL LINE}	ESC shift-BACK
	{DOWN}	ESC ctrl-equals
	{ESC}	ESC ESC
	{INS CHAR}	ESC ctrl-greater
	{INS LINE}	ESC shift-greater
	{LEFT}	ESC ctrl-plus
	{RIGHT}	ESC ctrl-star
	{SET TAB}	ESC shift-TAB
	{TAB}	ESC TAB
	{UP}	ESC ctrl-minus

---- Note ----

comma	is	,
equals	is	=
greater	is	>
less	is	<
minus	is	-
period	is	.
plus	is	+
semicolon	is	;
star	is	*



UPDATE FOR IRIDIS 2

The LOADFONT routine described on pages 46-47 needs the following corrections and clarifications.

1. Please note that LOADFONT is not a complete program. It is a subroutine that you can use with your own programs to load character fonts that were created with FONTEDIT.
2. On page 47, please change ENTER"D:LOADFONT" to ENTER"D:LOADFONT.LST"
3. Line 30700 can be improved by turning off the TRAP (that was set in line 30600). Here is the improved version:

```
30700 POKE 756,P:TRAP 50000:RETURN
```

(As you probably know, setting a trap to an illegal line number turns it off.)

Another question we get asked is "How can I change graphics modes, and not lose my custom font?" For example, to change to graphics mode 1:

```
100 T=PEEK(756):GRAPHICS 1:POKE 756,T
```



## ABOUT YOUR IRIDIS 2 TAPE

Your **IRIDIS 2** cassette tape has two Basic programs, three data files, and a Basic subroutine. With the Atari, it isn't easy to work with tapes with more than one file on them. Since **IRIDIS 2** has six files on one tape, for your convenience we suggest that you put each separate program or file on a separate tape. If you want to save a little money, you may want to put one thing at the beginning of each side of your tape. However, that means lots of rewinding, which is slow.

The **IRIDIS 2** tape has these programs and files, in this order:

<u>File:</u>	<u>Type:</u>	<u>Description:</u>
FONTEDIT.BAS	Program	Used to create character sets.
COMPUTER.FNT	Data File	A font we supply.
FANCY.FNT	Data File	Another font for FONTEDIT.
KNOTWORK.FNT	Data File	Font for the KNOTWORK program.
KNOTWORK.BAS	Program	The KNOTWORK program.
LOADFONT.LST	Subroutine	Basic code you need to load fonts.

Before we begin our step-by-step instructions, we want to emphasize that the files ending with '.FNT' are NOT Basic programs! They are files that contain sets of characters for use with FONTEDIT. If you try to use the CLOAD command with these data files, your Atari will complain.

Before we begin, take your IRIDIS master cassette and look at the back edge. You'll see two small tabs at either end. Take a ball-point pen and break out the tabs. This will make it impossible for you to accidentally record over your master!

1. Insert the IRIDIS 2 master tape in the Atari 410 recorder and press REWIND. When the tape is rewound, press STOP. Please note that this is the only time that you'll rewind the master tape during the following procedure.
2. Type: **CLOAD** and press Return. The Atari will beep once, reminding you to press PLAY on the recorder. After you have done that, press Return again. If you have the sound on your TV turned up, you'll be able to hear the machine loading the FONTEDIT program. When it's done, it will say READY on the TV screen.
3. Remove the master IRIDIS tape. Do not rewind it! We will want to read the next file on the tape in just a bit.
4. Label a blank cassette "FONTEDIT.BAS", and put it in the recorder. Type **CSAVE** and press Return. You'll hear two beeps, to remind you to press both RECORD and PLAY on the recorder. When you have done so, press Return and the Atari will save the FONTEDIT program on tape. When it says READY, rewind the tape and set it aside.
5. Put the IRIDIS master cassette back in the recorder.
6. Now type **RUN**. FONTEDIT will print its name in large letters, and wait for you to press Return. Next, it will print its normal display, and then the message: "Move Cursor With Joystick - Press Button When Ready". At this point, just press Return (because we don't want to edit a character right now). It will start blinking a dot in the upper left corner of the big picture of the current character. Now, press **L**, and it will ask: FONT TO LOAD FROM? Type: **C;**, (please note the colon) and press Return. You'll hear the beep telling you to press PLAY on the recorder. When you have done so, press Return again. Now the Atari will read data from cassette for the file called COMPUTER.FNT. You'll see two things - a small white line will jump around on the screen showing the font being read, and at the same time characters will be changing. When it's done (and it takes a while!) FONTEDIT will say: "Load Complete", and then start blinking one of the dots in the upper left corner again.



7. Remove the master tape. Do not rewind it!
8. Label a blank tape "COMPUTER.FNT", put it in the recorder and REWIND it. Now give the command **S** to tell FONTEDIT we want to Save a font. When it asks for a name, type **C:** and press Return. It will beep (twice), and you will press RECORD and PLAY on the recorder, and then Return again. When FONTEDIT is done saving the file it will tell you so, and then go back to its normal mode of blinking a dot.

Now, we have to do the same thing twice more as we load (from the IRIDIS master tape) and then save the next two font files on their own private cassettes.

9. Remove the tape that you used to save COMPUTER.FNT. Put the master IRIDIS tape back in the recorder.
10. Give the **L** command to load the next font file. Remember to type **C:** when it asks FONT TO LOAD FROM?, so that it will know to read from the cassette tape.
11. Take the master tape out of the recorder and put in a blank tape labelled "FANCY.FNT". Press **S** to give the Save command. Again, give the file name of **C:**.
12. Remove the tape that you used to record "FANCY.FNT" and put the master cassette back in the recorder. Type **L**, because we want to Load one more font file. When it asks for a name, type **C:**, and press PLAY on the recorder.
13. (Yawn...) Take the master tape out, and put in a tape labelled "KNOTWORK.FNT". Give the **S** command again. When it asks for a name, type: **C: and press RECORD and PLAY and then Return.**
14. Put the master tape back in the recorder. Give FONTEDIT the **Q** (as in "Quit") command. Type **CLOAD** to load the KNOTWORK program.
15. Take the master cassette out of the recorder, and put in a new tape labelled "KNOTWORK.BAS".
16. Type **CSAVE** to save KNOTWORK on its own tape.
17. (Almost done...) Put the master IRIDIS cassette back in the recorder.  
IMPORTANT NOTE: Type **NEW** to completely clear memory.

18. Type **ENTER "C:"** to bring in the LOADFONT subroutine. Note that the quotes are required.

19. Put a tape labelled "LOADFONT.LST Subroutine" in the recorder. Type **LIST "C:"** to save the file on tape. Again, the quotes are needed. (ENTER C: and LIST C: go together, just as CLOAD and CSAVE do. The difference is that ENTER and LIST work with untokenized programs. Also, ENTER does **NOT** clear memory, while CLOAD does. So, with ENTER you can easily add a subroutine such as LOADFONT.LST to your own programs.)

20. Done! Ain't computers fun?